
SmartThings Classic Developer Documentation

Release latest

SmartThings

April 10, 2019

I Latest Updates	1
1 July 07 2017	3
2 June 08 2017	5
3 May 04 2017	7
4 April 20 2017	9
5 March 22 2017	11
6 March 08 2017	13
7 March 02 2017	15
8 February 10 2017	17
9 February 08 2017	19
10 January 23 2017	21
11 January 03 2017	23
12 December 08 2016	25
13 November 30 2016	27
14 November 17 2016	29
15 November 15 2016	31
16 November 14 2016	33
17 November 10 2016	35
18 November 03 2016	37
19 October 26 2016	39
20 October 17 2016	41

21	October 13 2016	43
22	October 11 2016	45
23	October 06 2016	47
24	October 05 2016	49
25	September 23 2016	51
26	September 14 2016	53
27	September 09 2016	55
28	September 02 2016 (3)	57
29	September 02 2016 (2)	59
30	September 02 2016	61
31	August 17 2016	63
32	August 16 2016	65
33	August 15 2016	67
34	August 04 2016	69
35	July 28 2016	71
36	July 25 2016	73
37	July 21 2016	75
38	July 07 2016	77
39	June 23 2016	79
40	June 17 2016	81
41	June 13 2016	83
42	June 9 2016	85
43	May 27 2016	87
44	May 23 2016	89
II	Overview	91
45	Developer highlights	95
46	How it works	97
	46.1 SmartApps	97
	46.2 Device Handlers	97
47	An open platform	99

48 What's next	101
III Up and Running	103
49 Register	107
50 Explore	109
50.1 Account management	109
50.2 IDE and Simulator	109
51 Next steps	111
IV Groovy Basics	113
52 Overview	117
53 Installing Groovy	119
54 Optional semicolons	121
55 Comments	123
56 Objects	125
57 Optionally typed	127
58 Operators	129
59 Strings	131
60 Lists and Maps	133
61 Control structures	135
62 Calling methods	137
63 Getters and setters	139
64 Defining methods	141
65 Exception handling	143
66 Closures	145
67 Groovy truth	149
68 Default imports	151
69 What about classes?	153
70 Further reading	155
71 Next steps	157

V Groovy With SmartThings	159
72 How it works	163
73 Language simplifications	165
73.1 Classes and JARs	165
73.2 Restricted methods	165
73.3 Global variables	166
73.3.1 Constants	166
73.3.2 Mutable variables	166
73.4 Other notable restrictions	166
74 Allowed classes	167
75 Summary and next steps	171
VI Writing Your First SmartApp	173
76 Goals	177
77 Prerequisites	179
78 Create a SmartApp	181
79 Editor	183
80 Simulator	185
81 SmartApp basics	187
82 Definition	189
83 Preferences	191
83.1 Capabilities	192
84 Events and callback methods	193
85 Event Handler methods	195
86 Controlling devices	197
87 Using the Simulator	199
88 Publishing and installing	203
89 Turn off when motion inactive	207
90 Going further—adding flexibility	209
91 Complete code listing	213
92 How the switch turns on (or off)	215
93 Summary	217
94 Next steps	219
94.1 More about SmartApps	219

94.2 Fork it!	219
94.3 Device Handler development	219
VII Getting Help	221
95 Developer documentation	225
96 Community	227
97 SmartThings developer support	229
VIII Architecture	231
98 Big picture	235
98.1 Devices	235
98.2 Hub	236
98.3 Connectivity management	236
98.4 Device Handler execution	236
98.5 Subscription management	236
98.6 SmartApp execution	236
98.7 Web UI and IDE	237
99 Important concepts	239
99.1 Asynchronous and eventually consistent programming	239
99.2 Containers	239
99.3 Accounts	239
99.4 Locations and users	241
99.5 Groups	241
100 Capability taxonomy	243
100.1 Attributes and events	243
100.2 Commands	243
100.3 Custom capabilities	243
101 SmartThings cloud	245
102 Hubs and Locations	247
102.1 Consequences of sharding	247
IX Tools and IDE	249
103 Account Management	253
103.1 Locations	253
103.2 Hubs	253
103.3 Devices	254
103.4 SmartApps	254
103.5 Device Handlers	255
103.6 Publication requests	255
103.7 Live logging	255
104 Editor and Simulator	257
104.1 Creating a new SmartApp	257
104.2 Creating a new Device Handler	258

104.3 Using the editor	258
104.4 Using the Simulator	259
105 Logging	261
105.1 Overview	261
105.2 Logging levels	261
105.3 Logging exceptions	261
105.4 Logging examples	262
106 GitHub Integration	265
106.1 Overview	265
106.2 Setup	266
106.2.1 Step 1 - Enable GitHub integration	266
106.2.2 Step 2 - Connect your GitHub account to SmartThings	266
106.2.3 Step 3 - Create a fork	267
106.2.4 Step 4 - Clone the forked repository	268
106.2.5 Step 5 - Configure Git to sync fork with SmartThings	269
106.3 Repository structure	269
106.4 GitHub integration IDE tour	270
106.4.1 Color-coded names	270
106.4.2 GitHub actions buttons	270
Commit Changes	271
Update from Repo	271
Settings	271
106.5 How to	271
106.5.1 Add files from repository to the IDE	271
106.5.2 Get latest code from SmartThingsPublic repository	272
106.5.3 Commit changes in the IDE	273
106.5.4 Keep your cloned repo in sync with origin	273
106.6 Best practices	273
106.6.1 Sync with upstream repository frequently	273
106.7 FAQ	274
106.8 Getting help	274
X SmartApps	277
107 Anatomy and Life Cycle of a SmartApp	281
107.1 Types of SmartApps	281
107.1.1 Event Handler SmartApps	281
107.1.2 Solution Module SmartApps	281
107.1.3 Service Manager SmartApps	281
107.2 SmartApp structure	282
107.2.1 Definition	282
107.2.2 Preferences	282
107.2.3 Pre-defined callbacks	282
107.2.4 Event Handlers	283
107.3 SmartApp execution	283
107.4 Device preferences	283
107.5 Event subscriptions	284
107.6 SmartApp sandboxing	284
107.7 Execution location	284
108 Preferences and Settings	285
108.1 Preferences overview	285

108.2	Page definition	286
108.3	Section definition	287
108.4	Single preferences page	287
108.5	Multiple preferences pages	290
108.6	Preference elements and inputs	291
108.6.1	paragraph	291
108.6.2	icon	293
108.6.3	href	297
108.6.4	mode	300
108.6.5	label	302
108.6.6	app	304
108.6.7	input	304
108.6.8	Using device-specific inputs	306
108.7	Hide when empty	307
108.7.1	Working with other input types	308
108.8	Custom Remove button	308
108.9	Dynamic preferences	310
108.9.1	dynamicPage() options	312
108.10	Private settings	312
108.11	Examples	313
109	Storing Data With State	315
109.1	Quick example	315
109.2	State and Atomic State overview	316
109.3	Persistence model	316
109.4	How State works	317
109.5	State and potential race conditions	317
109.6	How Atomic State works	318
109.7	Choosing between State and Atomic State	318
109.8	What can be stored in State and Atomic State	319
109.8.1	Supported types	319
109.8.2	Other object types	320
109.9	Working with the <code>state</code> object	320
109.9.1	Adding values	321
109.9.2	Retrieving values	321
109.9.3	Updating values	321
109.9.4	Removing values	321
109.9.5	Iterating over <code>state</code>	321
109.9.6	Working with collections	322
109.10	Working with the <code>atomicState</code> object	322
109.10.1	Adding values	322
109.10.2	Updating values	322
109.10.3	Removing values	323
109.10.4	Iterating over all values	323
109.10.5	Working with collections	323
109.11	Storage size limits	323
109.12	State in parent-child relationships	324
109.13	Summary	324
110	Events and Subscriptions	325
110.1	Subscribe to specific device Events	325
110.2	Subscribe to all device Events	326
110.3	Subscribe to multiple devices	326
110.4	Subscribe to Location Events	326

110.5 The Event object	327
110.6 See also	327
111 Working with Devices	329
111.1 Device overview	329
111.2 Preferences—selecting the devices	329
111.3 Interacting with devices	330
111.4 Device attributes	330
111.5 Device commands	330
111.6 Getting device current values	330
111.7 Querying event history	331
111.8 Sending commands	332
111.9 Interacting with multiple devices	332
111.10 See also	333
112 Modes	335
112.1 Overview	335
112.2 Getting the current Mode	335
112.3 Getting all Modes	335
112.4 Setting the Mode	336
112.5 Allowing users to select Modes	336
112.6 Mode events	336
112.7 Example	337
112.8 Further reading	337
113 Routines	339
113.1 Overview	340
113.2 Get available Routines	341
113.3 Execute Routines	341
113.4 Allowing users to select Routines	341
113.5 Routine Events	342
113.6 Example	342
113.7 Further reading	344
114 Scheduling	345
114.1 Overview	345
114.2 Schedule from now— <code>runIn()</code>	345
114.3 Run once in the future— <code>runOnce()</code>	346
114.4 Run on a recurring schedule	347
114.4.1 Schedule once per day	347
114.4.2 Schedule every X minutes or hours	348
114.4.3 Schedule using cron	349
114.5 Passing data to the handler method	350
114.6 Removing scheduled executions	351
114.7 Viewing schedules in the IDE	352
114.8 Best practices	352
114.8.1 Avoid chained <code>runIn()</code> calls	353
114.8.2 Prefer <code>runEvery*()</code> over cron	353
114.8.3 Execution time may not be in exact seconds	353
114.8.4 Do not aggressively schedule	353
114.8.5 <code>unschedule()</code> is expensive	353
114.8.6 Number of scheduled executions limit	354
114.9 Examples	354
115 Working With Time	355

115.1 Taking action within a time window	355
115.2 Execute only on certain days	356
115.3 Working with time zones	357
116Sunset and Sunrise	359
116.1 Sunrise and sunset Events	359
116.1.1 Taking action at sunrise or sunset	359
116.1.2 Taking action before or after	359
116.2 Looking up sunrise or sunset directly	361
116.3 Polling for sunrise or sunset	361
116.4 Examples	361
117App Touch	363
117.1 Subscribe to app	363
118Making Synchronous External HTTP Requests	365
118.1 HTTP methods	365
118.2 Configuring the request	366
118.3 Handling the response	366
118.4 Host and timeout limitations	368
118.4.1 Host and IP address restrictions	368
118.4.2 Request timeout limit	368
118.5 Try it out	368
118.6 See also	369
119Making Asynchronous External HTTP Requests (Beta)	371
119.1 Overview	371
119.2 Quick example	372
119.3 Synchronous versus asynchronous	372
119.4 The include Statement	374
119.5 Configuring the request	374
119.5.1 URI and path	375
119.5.2 Request headers	375
119.5.3 Query parameters	375
119.5.4 Request body	376
119.6 Handling the response	377
119.6.1 Response status code	377
119.6.2 Response headers	378
119.6.3 Error responses	378
119.6.4 JSON responses	379
119.6.5 XML responses	379
119.6.6 Getting the raw response	380
119.7 Passing data to the request handler	380
119.8 Available methods	381
119.9 Host, timeout, response, and data size limits	381
119.9.1 Host and IP address restrictions	381
119.9.2 Request timeout limit	381
119.9.3 Response size limit	382
119.9.4 Data size limit	382
119.10Using asynchronous HTTP in parent-child relationships	382
119.11When to use asynchronous HTTP requests	382
119.12Refactoring to asynchronous HTTP requests	382
119.12.1Find high-value opportunities	382
119.12.2Refactoring strategies	383
119.13Example	384

119.14	Related documentation	384
120	Sending Notifications	385
120.1	Send notifications with Contact Book	385
120.1.1	Selecting Contacts to notify	385
120.1.2	Send notifications to Contacts	386
120.1.3	Handling disabled Contact Book	387
120.1.4	Complete example	387
120.2	Send push notifications	388
120.3	Send SMS notifications	389
120.4	Send both push and SMS notifications	390
120.5	Only display message in the notifications feed	391
120.6	Examples	391
120.7	Related API documentation	391
121	Parent-Child SmartApps	393
121.1	Overview	393
121.2	The parent SmartApp	394
121.3	The child SmartApp	394
121.4	Communicating between parent and children	395
121.5	Preventing more than one parent instance	395
121.6	Example	395
121.7	Tips and best practices	400
121.8	Summary	401
122	Example: Bon Voyage	403
122.1	Bon Voyage	403
122.2	SmartApp preferences	403
122.3	Monitor and react	404
122.4	Related documentation	407
122.5	Complete source code	407
XI	Web Services SmartApps	409
123	Web Services SmartApps Overview	413
123.1	Introduction	413
123.2	Concepts	413
123.3	How it works	414
123.3.1	OAuth-integrated app installation flow	414
123.4	The end-user journey	415
123.4.1	Initiate connection from external system	415
123.4.2	Authentication and authorization	415
123.4.3	Application configuration	416
124	Web Services Tutorial—SmartApp	419
124.1	Overview	419
124.2	Create a new SmartApp	419
124.3	Define preferences	419
124.4	Specify endpoints	420
124.5	GET switch information	421
124.6	UPDATE the switches	421
124.7	Self-publish the SmartApp	422
124.8	Run the SmartApp in the Simulator	422
124.9	Make API calls to the SmartApp	422

124.10	Uninstall the SmartApp	423
124.11	Summary	423
125	Web Services SmartApp Tutorial–Authorization Flow	425
125.1	Overview	425
125.2	Prerequisites	426
125.3	Bootstrap the Sinatra app	426
125.4	Get an authorization code	427
125.5	Get an access token	429
125.6	Discover the endpoint	429
125.7	Make API calls	430
125.8	Summary	431
126	The SmartApp	433
126.1	Enable OAuth	433
126.2	Preferences	434
126.3	Mapping endpoints	435
126.4	Request handling	436
126.4.1	Path variables	436
126.4.2	Query parameters	437
126.4.3	Request body parameters	437
126.5	Response handling	438
126.5.1	Defaults	438
126.5.2	Automatic JSON serialization	438
126.5.3	Using render () to control the response	438
126.6	Error handling	439
126.6.1	Default errors	439
126.6.2	Custom errors	440
127	Authorization	441
127.1	Overview	441
127.2	Get authorization code	441
127.3	Get access token	442
127.4	Get SmartApp endpoints	443
127.5	Make REST calls	443
128	Troubleshooting	445
128.1	General	445
128.2	Errors during installation	445
128.2.1	“<clientID> is not associated with a SmartApp in Location” after selecting Location	445
128.2.2	“Please select at least one device to authorize” error after clicking Authorize	446
XII	Device Handlers	447
129	Quick Start	451
129.1	Create a new Device Handler	451
129.2	Create a Virtual Device	453
129.3	Test your Device Handler with Virtual Device	454
129.4	Next steps	454
130	Overview	459
130.1	Core concepts	461
130.1.1	Capabilities	461
130.1.2	Commands	461

130.1.3 Attributes	462
130.1.4 Actuator and Sensor	462
130.2 Protocols	462
130.3 Execution location	462
131 Simulator	463
131.1 Overview	463
131.2 Status	463
131.3 Reply	465
131.4 Summary	465
132 Definition	467
132.1 Capabilities	467
132.2 Attributes	468
132.3 Commands	468
132.4 Fingerprinting	469
132.4.1 ZigBee fingerprinting	469
132.4.2 Z-Wave fingerprinting	469
Z-Wave raw description	469
New Z-Wave fingerprint format	470
Legacy Z-Wave fingerprint format	471
132.4.3 Fingerprinting best practices	471
Add multiple fingerprints	471
Device pairing process	471
Overly general fingerprints	472
133 Tiles	473
133.1 Overview	473
133.2 Tiles basics	475
133.2.1 Main and details tiles configuration	476
133.2.2 Grid layout	477
133.2.3 Tile size	478
133.2.4 Allowing the user to change the icon	478
133.3 Tiles and Attribute state	479
133.3.1 State actions	479
133.3.2 Transition states	480
133.3.3 State labels	480
133.3.4 Background color	481
133.3.5 State selection algorithm	482
133.3.6 Icons	482
133.4 Single-Attribute Tiles	482
133.4.1 Standard Tile	482
133.4.2 Value Tile	484
133.4.3 Slider Control Tile	485
133.4.4 Color Control Tile	485
133.4.5 Carousel Tile	486
133.5 Multi-Attribute Tiles	486
133.5.1 Basics	487
133.5.2 Multi-Attribute Tile types	487
133.5.3 Attribute state and control keys	487
133.5.4 Lighting Multi-Attribute Tile	488
133.5.5 Thermostat Multi-Attribute Tile	489
133.5.6 Multimedia Multi-Attribute Tile	491
133.5.7 Generic Multi-Attribute Tile	492

133.5.8 Controls summary	494
133.6 Color standards	495
133.6.1 Colors	495
133.6.2 Examples	496
133.7 Additional information	496
133.8 Examples	497
134 Preferences	499
134.1 Overview	499
134.2 Defining preferences	499
134.3 Device preferences are flat	499
134.4 Display on setup	500
134.5 Supported input types	500
134.6 Getting preference input values	500
134.7 Example	501
134.8 Additional notes	501
135 Parse and Events	503
135.1 Overview	503
135.2 Parse, Events, and Attributes	504
135.2.1 Creating Events	504
135.2.2 Multiple Events	504
135.2.3 Generating Events outside of parse	505
135.3 Tips	505
136 Z-Wave Primer	507
136.1 Command classes	507
136.2 Listening and sleepy devices	508
136.3 Configuration	508
136.4 Association	509
137 Building Z-Wave Device Handlers	511
137.1 Parsing Events	511
137.2 Sending commands	512
137.3 Sending commands in response to Events	513
138 Z-Wave Example	515
139 ZigBee Primer	523
139.1 Device Network ID	523
139.2 Endpoints	524
139.3 Clusters	524
139.4 Commands	524
139.5 Read and Write Attributes	525
139.6 Configure reporting	525
139.7 Device discovery	525
139.8 Useful ZigBee references	526
140 Building ZigBee Device Handlers	527
140.1 Commands	527
140.1.1 Read	527
140.1.2 Write	527
140.1.3 Command	528
140.1.4 Configure	528
140.2 ZigBee utilities	529

140.3	Best practices	529
140.4	Using the ZigBee Device Form	529
140.4.1	What it does	530
140.4.2	Use it if	530
140.4.3	How to use	531
141	ZigBee Example	533
142	Other Useful Methods	535
142.1	Scheduling	535
142.2	Storing data	535
142.3	Making external HTTP requests	535
143	Device Certification Overview	537
XIII	Cloud- and LAN-connected Devices	539
144	Service Manager Design Pattern	543
144.1	Basic overview	543
144.2	Cloud-connected devices	543
144.3	LAN-connected devices	543
145	Building Cloud-connected Device Types	545
145.1	Division of Labor	545
145.1.1	Service Manager responsibilities	545
145.1.2	Device Handler responsibilities	545
145.1.3	How it all works	545
145.2	Building the Service Manager	546
145.2.1	Authentication using OAuth	546
	End user experience	547
	Implementation	551
	Initialize endpoint	552
	Callback endpoint	553
	Refreshing the OAuth token	555
145.2.2	Discovery	556
	Identifying devices in the third-party device cloud	556
	Creating child devices	556
	Getting initial device state	556
145.2.3	Handling adds, changes, deletes	557
	singleInstance Service Manager	557
	Implicit creation of new child Devices	557
	Implicit removal of child Devices	558
	Changes in Device name	558
	Explicit delete actions	558
145.3	Building the Device Handler	558
145.3.1	The Parse method	558
145.3.2	Sending commands to the third-party cloud	559
145.3.3	Receiving Events from the third-party cloud	559
145.3.4	Generating Events at the request of the Service Manager	560
146	Building LAN-connected Device Types	561
146.1	Division of Labor	561
146.1.1	Service Manager responsibilities	561
146.1.2	Device Handler responsibilities	561

146.1.3	How it all works	562
146.2	Building the Service Manager	562
146.2.1	Discovery	563
146.2.2	Verification	564
146.2.3	Inclusion	565
146.2.4	Health	565
146.2.5	Best practices	566
146.2.6	References and resources	567
146.3	Building the Device Type	567
146.3.1	Making outbound HTTP calls with HubAction	567
146.3.2	Overview	567
146.3.3	Creating a HubAction object	567
146.3.4	Parsing the response	568
146.3.5	Getting the addresses	568
146.3.6	Wake on LAN (WOL)	569
146.3.7	REST requests	570
146.3.8	UPnP/SOAP requests	570
146.3.9	Subscribing to device Events	570
146.3.10	References and resources	571
147	Automatic LAN Device Discovery	573
147.1	Impact on the developer	573
147.1.1	Supported LAN-connected Devices	573
148	Capturing and Displaying Camera Pictures	575
148.1	Image Capture Capability	575
148.2	Tiles for taking and viewing pictures	575
148.3	Capture and display images	576
148.3.1	LAN-connected cameras	576
148.3.2	Cloud-connected cameras	577
148.4	Retrieving an image	578
148.5	Image size limits	579
148.6	Allowed image name characters	579
148.7	Image storage duration	579
148.8	Supported image formats	579
148.9	Related documentation	580
XIV	Composite Devices	581
149	Device Handler for a Composite Device	585
149.1	Parent Device Handler	585
149.2	Child Device Handler	586
150	Deleting a Composite Device	587
151	Composite Device Tiles	589
151.1	Example: Simulated refrigerator	590
151.2	Example composite tile code	594
XV	Arduino ThingShield	595
152	Installing the library	599

153	Pairing the shield	601
154	Changing the Device Handler	603
155	Arduino examples	605
XVI	Rate Limits	607
156	SmartApp and Device Handler rate limits	611
156.1	Execution count limits	611
156.2	Execution time limits	611
157	Web services rate limit headers	613
158	SMS rate limits	615
159	Parent-child relationship limit	617
160	Avoiding rate limits	619
XVII	Publishing Code	621
161	For yourself	625
161.1	Ensure proper Location	625
161.2	Publish	626
162	For public distribution	627
162.1	Review process	627
XVIII	Code Review Guidelines and Best Practices	629
163	General	633
163.1	Code should be readable	633
163.2	Don't repeat yourself	633
163.3	Methods should serve a single purpose	633
163.4	Do not submit unused code	633
163.5	Do not use offensive, profane, or libelous language	633
163.6	Comment appropriately	634
163.7	Handle all <code>if()</code> and <code>switch()</code> cases	635
163.8	Verify assumptions	635
163.9	Use consistent return values	635
163.10	Be careful indexing into arrays	635
163.11	Use the Elvis operator correctly	636
163.12	Handle null values	636
163.13	Use Groovy truth correctly	637
164	Using State	639
164.1	<code>state</code> is not an unbounded database	639
164.2	Understand how <code>state</code> works	639
164.3	Understand when to use <code>atomicState</code> vs. <code>state</code>	639
164.4	Take care when storing collections in <code>atomicState</code>	639

165	Web Services	641
165.1	Document external HTTP requests	641
165.2	Document any exposed endpoints	641
166	Scheduling	643
166.1	Avoid recurring short schedules	643
166.2	Avoid chained <code>runIn()</code> calls	643
167	Security considerations	645
167.1	Subscriptions should be clear	645
167.2	Subscriptions should be specific	645
167.3	Do not use dynamic method execution	645
167.4	Do not hard-code SMS messages	646
168	Performance	647
168.1	Do not use busy loops	647
168.2	Do not use <code>synchronized()</code>	647
169	LAN-specific	649
169.1	Use the device-specific search	649
169.2	Handle IP change	649
170	Parent-child relationships	651
170.1	Use separate files	651
XIX	Capabilities Reference	653
171	Introduction	657
172	Data Types	659
173	Acceleration Sensor	661
173.1	Definition	661
174	Actuator	663
174.1	Definition	663
175	Air Conditioner Mode	665
175.1	Definition	665
176	Air Quality Sensor	667
176.1	Definition	667
177	Alarm	669
177.1	Definition	669
178	Audio Mute	671
178.1	Definition	671
179	Audio Notification	673
179.1	Definition	673
180	Audio Track Data	675
180.1	Definition	675

181	Audio Volume	677
181.1	Definition	677
182	Battery	679
182.1	Definition	679
183	Beacon	681
183.1	Definition	681
184	Bridge	683
184.1	Definition	683
185	Bulb	685
185.1	Definition	685
186	Button	687
186.1	Definition	687
187	Carbon Dioxide Measurement	689
187.1	Definition	689
188	Carbon Monoxide Detector	691
188.1	Definition	691
189	Color Control	693
189.1	Definition	693
190	Color Temperature	695
190.1	Definition	695
191	Color	697
191.1	Definition	697
192	Color Mode	699
192.1	Definition	699
193	Configuration	701
193.1	Definition	701
194	Consumable	703
194.1	Definition	703
195	Contact Sensor	705
195.1	Definition	705
196	Demand Response Load Control	707
196.1	Definition	707
197	Dishwasher Mode	709
197.1	Definition	709
198	Dishwasher Operating State	711
198.1	Definition	711
199	Door Control	713
199.1	Definition	713

200Dryer Mode	715
200.1 Definition	715
201Dryer Operating State	717
201.1 Definition	717
202Dust Sensor	719
202.1 Definition	719
203Energy Meter	721
203.1 Definition	721
204Estimated Time Of Arrival	723
204.1 Definition	723
205Execute	725
205.1 Definition	725
206Fan Speed	727
206.1 Definition	727
207Filter Status	729
207.1 Definition	729
208Garage Door Control	731
208.1 Definition	731
209Geolocation	733
209.1 Definition	733
210Holdable Button	735
210.1 Definition	735
211Illuminance Measurement	737
211.1 Definition	737
212Image Capture	739
212.1 Definition	739
213Indicator	741
213.1 Definition	741
214Infrared Level	743
214.1 Definition	743
215Light	745
215.1 Definition	745
216Lock Only	747
216.1 Definition	747
217Lock	749
217.1 Definition	749
218Media Controller	751
218.1 Definition	751

219Media Input Source	753
219.1 Definition	753
220Media Playback Repeat	755
220.1 Definition	755
221Media Playback Shuffle	757
221.1 Definition	757
222Media Playback	759
222.1 Definition	759
223Media Presets	761
223.1 Definition	761
224Media Track Control	763
224.1 Definition	763
225Momentary	765
225.1 Definition	765
226Motion Sensor	767
226.1 Definition	767
227Music Player	769
227.1 Definition	769
228Notification	773
228.1 Definition	773
229Odor Sensor	775
229.1 Definition	775
230Outlet	777
230.1 Definition	777
231Oven Mode	779
231.1 Definition	779
232Oven Operating State	781
232.1 Definition	781
233Oven Setpoint	785
233.1 Definition	785
234pH Measurement	787
234.1 Definition	787
235Polling	789
235.1 Definition	789
236Power Consumption Report	791
236.1 Definition	791
237Power Meter	793
237.1 Definition	793

238Power Source	795
238.1 Definition	795
239Presence Sensor	797
239.1 Definition	797
240Rapid Cooling	799
240.1 Definition	799
241Refresh	801
241.1 Definition	801
242Refrigeration Setpoint	803
242.1 Definition	803
243Relative Humidity Measurement	805
243.1 Definition	805
244Relay Switch	807
244.1 Definition	807
245Robot Cleaner Cleaning Mode	809
245.1 Definition	809
246Robot Cleaner Movement	811
246.1 Definition	811
247Robot Cleaner Turbo Mode	813
247.1 Definition	813
248Sensor	815
248.1 Definition	815
249Shock Sensor	817
249.1 Definition	817
250Signal Strength	819
250.1 Definition	819
251Sleep Sensor	821
251.1 Definition	821
252Smoke Detector	823
252.1 Definition	823
253Sound Pressure Level	825
253.1 Definition	825
254Sound Sensor	827
254.1 Definition	827
255Speech Recognition	829
255.1 Definition	829
256Speech Synthesis	831
256.1 Definition	831

257	Step Sensor	833
257.1	Definition	833
258	Switch Level	835
258.1	Definition	835
259	Switch	837
259.1	Definition	837
260	Tamper Alert	839
260.1	Definition	839
261	Temperature Measurement	841
261.1	Definition	841
262	Thermostat Cooling Setpoint	843
262.1	Definition	843
263	Thermostat Fan Mode	845
263.1	Definition	845
264	Thermostat Heating Setpoint	847
264.1	Definition	847
265	Thermostat Mode	849
265.1	Definition	849
266	Thermostat Operating State	851
266.1	Definition	851
267	Thermostat Setpoint	853
267.1	Definition	853
268	Thermostat	855
268.1	Definition	855
269	Three Axis	861
269.1	Definition	861
270	Timed Session	863
270.1	Definition	863
271	Tone	865
271.1	Definition	865
272	Touch Sensor	867
272.1	Definition	867
273	Tv Channel	869
273.1	Definition	869
274	Ultraviolet Index	871
274.1	Definition	871
275	Valve	873
275.1	Definition	873

276	Video Clips	875
276.1	Definition	875
277	Video Stream	877
277.1	Definition	877
278	Voltage Measurement	879
278.1	Definition	879
279	Washer Mode	881
279.1	Definition	881
280	Washer Operating State	883
280.1	Definition	883
281	Water Sensor	887
281.1	Definition	887
282	Window Shade	889
282.1	Definition	889
XX	API Documentation	891
283	How to read the docs	895
283.1	Objects	895
283.2	Object wrappers	895
283.3	Dynamic methods	895
283.4	Conventions	896
284	API Contents	897
284.1	SmartApp	897
284.1.1	installed()	897
284.1.2	updated()	898
284.1.3	uninstalled()	898
284.1.4	<device or capability preference name>	898
284.1.5	<number or decimal preference name>	899
284.1.6	<text, mode, or time preference name>	899
284.1.7	addChildApp()	900
284.1.8	addChildDevice()	900
284.1.9	apiServerUrl()	901
284.1.10	atomicState	901
284.1.11	canSchedule()	902
284.1.12	createAccessToken()	902
284.1.13	findAllChildAppsByName()	902
284.1.14	findAllChildAppsByNamespaceAndName()	903
284.1.15	findChildAppByName()	903
284.1.16	findChildAppByNamespaceAndName()	904
284.1.17	getAllChildApps()	904
284.1.18	getChildApps()	904
284.1.19	deleteChildDevice()	905
284.1.20	getAllChildDevices()	905
284.1.21	getApiServerUrl()	905
284.1.22	getChildDevice()	905
284.1.23	getChildDevices()	905

284.1.24getColorUtil()	906
284.1.25getLocation()	906
284.1.26getSunriseAndSunset()	906
284.1.27getTwcConditions()	907
284.1.28getTwcForecast()	908
284.1.29getTwcLocation()	915
284.1.30getTwcAlerts()	916
284.1.31getTwcAlertDetail()	918
284.1.32getWeatherFeature() - Deprecated	920
284.1.33httpDelete()	920
284.1.34httpError()	921
284.1.35httpGet()	921
284.1.36httpHead()	922
284.1.37httpPost()	923
284.1.38httpPostJson()	923
284.1.39httpPut()	924
284.1.40httpPutJson()	925
284.1.41nextOccurrence()	926
284.1.42now()	926
284.1.43parseJson()	926
284.1.44parseXml()	927
284.1.45parseLanMessage()	927
284.1.46parseSoapMessage()	927
284.1.47render()	927
284.1.48revokeAccessToken()	928
284.1.49runIn()	928
284.1.50runEvery1Minute()	929
284.1.51runEvery5Minutes()	930
284.1.52runEvery10Minutes()	931
284.1.53runEvery15Minutes()	931
284.1.54runEvery30Minutes()	932
284.1.55runEvery1Hour()	933
284.1.56runEvery3Hours()	933
284.1.57runOnce()	934
284.1.58schedule()	935
284.1.59sendEvent()	936
284.1.60sendHubCommand()	936
284.1.61sendLocationEvent()	937
284.1.62sendNotification()	938
284.1.63sendNotificationEvent()	938
284.1.64sendNotificationToContacts()	938
284.1.65sendPush()	939
284.1.66sendPushMessage()	939
284.1.67sendSms()	940
284.1.68sendSmsMessage()	940
284.1.69setLocationMode()	940
284.1.70settings	941
284.1.71state	941
284.1.72stringToMap()	942
284.1.73subscribe()	942
284.1.74subscribeToCommand()	943
284.1.75timeOfDayIsBetween()	943
284.1.76timeOffset()	944
284.1.77timeToday()	944

284.1.78	timeTodayAfter()	945
284.1.79	timeZone()	946
284.1.80	toDateTime()	946
284.1.81	unsubscribe()	946
284.1.82	unsubscribe()	947
284.2	Device Handler	947
284.2.1	<command name>()	948
284.2.2	parse()	948
284.2.3	addChildDevice()	949
284.2.4	apiServerUrl()	951
284.2.5	attribute()	951
284.2.6	capability()	952
284.2.7	carouselTile()	952
284.2.8	childDeviceTile()	953
284.2.9	command()	954
284.2.10	controlTile()	955
284.2.11	createEvent()	955
284.2.12	definition()	956
284.2.13	details()	957
284.2.14	device	957
284.2.15	fingerprint()	958
284.2.16	getApiServerUrl()	958
284.2.17	getChildDevices()	958
284.2.18	getColorUtil()	959
284.2.19	getImage()	959
284.2.20	httpDelete()	959
284.2.21	httpGet()	960
284.2.22	httpHead()	960
284.2.23	httpPost()	961
284.2.24	httpPostJson()	962
284.2.25	httpPut()	962
284.2.26	httpPutJson()	963
284.2.27	main()	964
284.2.28	metadata()	964
284.2.29	reply()	965
284.2.30	runEvery1Minute()	966
284.2.31	runEvery5Minutes()	966
284.2.32	runEvery10Minutes()	967
284.2.33	runEvery15Minutes()	968
284.2.34	runEvery30Minutes()	968
284.2.35	runEvery1Hour()	969
284.2.36	runEvery3Hours()	969
284.2.37	runIn()	970
284.2.38	runOnce()	971
284.2.39	schedule()	971
284.2.40	sendEvent()	972
284.2.41	simulator()	973
284.2.42	standardTile()	974
284.2.43	state	974
284.2.44	state()	975
284.2.45	status()	976
284.2.46	storeImage()	976
284.2.47	storeTemporaryImage()	977
284.2.48	tiles()	979

284.2.49	valueTile()	979
284.2.50	zigbee	980
284.2.51	zwave	980
284.3	AppState	981
284.3.1	getDateValue()	981
284.3.2	getId()	981
284.3.3	getDescriptionText()	982
284.3.4	getDoubleValue()	982
284.3.5	getFloatValue()	983
284.3.6	getIntegerValue()	983
284.3.7	getIsoDate()	984
284.3.8	getJsonValue()	984
284.3.9	getLastUpdated()	984
284.3.10	getLongValue()	985
284.3.11	getName()	985
284.3.12	getNumberValue()	985
284.3.13	getNumericValue()	986
284.3.14	getUnit()	986
284.3.15	getValue()	987
284.3.16	getXyzValue()	987
284.4	Async HTTP API (Beta)	987
284.4.1	delete()	988
284.4.2	get()	989
284.4.3	head()	990
284.4.4	patch()	990
284.4.5	post()	991
284.4.6	put()	992
284.5	AsyncResponse (Beta)	993
284.5.1	getData()	994
284.5.2	getErrorData()	994
284.5.3	getErrorJson()	994
284.5.4	getErrorMessage()	995
284.5.5	getErrorXml()	995
284.5.6	getHeaders()	995
284.5.7	getJson()	996
284.5.8	getStatus()	996
284.5.9	getWarningMessages()	996
284.5.10	getXml()	997
284.5.11	hasError()	997
284.6	Attribute	997
284.6.1	getDataType()	998
284.6.2	getName()	998
284.6.3	getValues()	999
284.7	Capability	999
284.7.1	getAttributes()	999
284.7.2	getCommands()	1000
284.7.3	getName()	1000
284.8	ColorUtilities	1001
284.8.1	hexToRgb()	1001
284.8.2	rgbToHex()	1002
284.9	Command	1002
284.9.1	getArguments()	1002
284.9.2	getName()	1003
284.10	Device	1003

284.10.1<attribute name>State	1004
284.10.2<command name>()	1004
284.10.3current<Uppercase attribute name>	1005
284.10.4currentState()	1006
284.10.5currentValue()	1006
284.10.6events()	1007
284.10.7eventsBetween()	1007
284.10.8eventsSince()	1008
284.10.9getCapabilities()	1008
284.10.10getDeviceNetworkId()	1009
284.10.11getDisplayName()	1009
284.10.12getHub()	1009
284.10.13getId()	1009
284.10.14getLabel()	1010
284.10.15getLastActivity()	1010
284.10.16getManufacturerName()	1010
284.10.17getModelName()	1010
284.10.18getStatus()	1011
284.10.19getName()	1011
284.10.20getSupportedAttributes()	1011
284.10.21getSupportedCommands()	1012
284.10.22getTypeName()	1012
284.10.23hasAttribute()	1012
284.10.24hasCapability()	1013
284.10.25hasCommand()	1014
284.10.26latestState()	1014
284.10.27latestValue()	1015
284.10.28statesBetween()	1015
284.10.29statesSince()	1016
284.11Event	1017
284.11.1getData()	1017
284.11.2getDate()	1017
284.11.3getDateValue()	1018
284.11.4getDescription()	1018
284.11.5getDescriptionText()	1018
284.11.6getDevice()	1018
284.11.7getDisplayName()	1019
284.11.8getDeviceId()	1019
284.11.9getId()	1019
284.11.10getDoubleValue()	1019
284.11.11getFloatValue()	1020
284.11.12getHubId()	1020
284.11.13getInstalledSmartAppId()	1021
284.11.14getIntegerValue()	1021
284.11.15getIsoDate()	1021
284.11.16getJsonValue()	1022
284.11.17getLinkText()	1022
284.11.18getLocation()	1022
284.11.19getLocationId()	1022
284.11.20getLongValue()	1023
284.11.21getName()	1023
284.11.22getNumberValue()	1023
284.11.23getNumericValue()	1024
284.11.24getSource()	1024

284.11.2	getStringValue()	1025
284.11.2	getUnit()	1025
284.11.2	getValue()	1025
284.11.2	getXyzValue()	1026
284.11.2	isDigital()	1026
284.11.3	isPhysical()	1026
284.11.3	isStateChange()	1027
284.12	Hub	1027
284.12.1	getFirmwareVersionString()	1027
284.12.2	getId()	1028
284.12.3	getLocalIP()	1028
284.12.4	getLocalSrvPortTCP()	1028
284.12.5	getName()	1028
284.12.6	getType()	1029
284.12.7	getZigbeeEui()	1029
284.12.8	getZigbeeId()	1029
284.13	HubAction	1029
284.14	InstalledSmartApp	1031
284.14.1	currentState()	1031
284.14.2	getAccountId()	1031
284.14.3	getAllChildApps()	1032
284.14.4	getAppSettings()	1032
284.14.5	getChildApps()	1032
284.14.6	getChildDevices()	1033
284.14.7	getExecutionIsModeRestricted()	1033
284.14.8	getExecutableModes()	1033
284.14.9	getId()	1033
284.14.10	getInstallationState()	1033
284.14.11	getLabel()	1034
284.14.12	getName()	1034
284.14.13	getNamespace()	1034
284.14.14	getParent()	1034
284.14.15	getSubscriptions()	1034
284.14.16	statesBetween()	1035
284.14.17	statesSince()	1035
284.14.18	updateLabel()	1036
284.15	Location	1036
284.15.1	getContactBookEnabled()	1036
284.15.2	getCurrentMode()	1036
284.15.3	getId()	1037
284.15.4	getHubs()	1037
284.15.5	getLatitude()	1037
284.15.6	getLongitude()	1037
284.15.7	getMode()	1038
284.15.8	getModes()	1038
284.15.9	getName()	1038
284.15.10	getMode()	1038
284.15.11	getTemperatureScale()	1039
284.15.12	getTimeZone()	1039
284.15.13	getZipCode()	1039
284.16	Mode	1039
284.16.1	getId()	1040
284.16.2	getName()	1040
284.17	State	1040

284.17.1	getDate()	1041
284.17.2	getDateValue()	1041
284.17.3	getDoubleValue()	1041
284.17.4	getFloatValue()	1041
284.17.5	getId()	1042
284.17.6	getIntegerValue()	1042
284.17.7	getIsoDate()	1043
284.17.8	getJsonValue()	1043
284.17.9	getLongValue()	1043
284.17.10	getName()	1044
284.17.11	getNumberValue()	1044
284.17.12	getNumericValue()	1044
284.17.13	getStringValue()	1045
284.17.14	getUnit()	1045
284.17.15	getValue()	1045
284.17.16	getXyzValue()	1045
284.18	ZigBee Reference	1046
284.18.1	Parse methods	1047
	zigbee.getEvent()	1047
284.18.2	Low level commands	1048
	additionalParams	1048
	zigbee.command()	1048
	zigbee.readAttribute()	1049
	zigbee.writeAttribute()	1049
	zigbee.configureReporting()	1050
284.18.3	ZigBee Capabilities	1050
	zigbee.on()	1051
	zigbee.off()	1051
	zigbee.setLevel()	1052
	zigbee.setColorTemperature()	1052
284.18.4	ZigBee helper commands	1052
	zigbee.parseDescriptionAsMap()	1052
	zigbee.convertToHexString()	1052
	zigbee.convertHexToInt()	1053
	zigbee.hexNotEqual()	1053
	zigbee.parseZoneStatus()	1053
284.18.5	Additional ZigBee classes	1053
	ZoneStatus	1053
	Accessing a Property/attribute	1054
	DataType	1056
	DataType constants	1056
	DataType.getLength()	1058
	DataType.isVariableLength()	1058
	DataType.isDiscrete()	1058
	DataType.pack()	1058
284.19	Z-Wave Reference	1059

XXI Contributing to the Docs

1061

285 Writing Style Guide

1063

285.1	Titles and headings	1063
285.1.1	Avoid framing as questions	1063
285.1.2	Avoid italics (emphasis)	1064

285.1.3	Document titles	1064
285.1.4	What not to capitalize in title	1064
285.1.5	What to capitalize in title	1064
285.1.6	Section headings	1064
285.2	UI elements	1065
285.3	List elements	1065
285.4	Page structure	1066
285.4.1	Page title	1067
285.4.2	Headings	1067
285.5	reStructuredText syntax	1067
285.5.1	Links	1067
285.5.2	Lists	1068
285.5.3	Inline markup	1068
285.5.4	Code examples	1068
285.5.5	Images	1069
285.5.6	Admonitions	1069
285.5.7	Tables	1070
285.6	API reference documents	1071
285.6.1	Organization	1071
285.6.2	Introduction	1071
285.6.3	Method documentation	1072
	Signature	1072
	Parameters	1072
	Returns	1073
	Throws	1073
	Example	1073
285.7	Miscellaneous tips	1073
285.8	SmartThings glossary	1074
285.9	Further reading	1074

Part I

Latest Updates

July 07 2017

Changes to the thermostatCoolingSetpoint Capability:

- `coolingSetpointMin` and `coolingSetpoingMax` attributes replaced with `coolingSetpointRange`.

Changes to the thermostatFanMode Capability:

- `supportedThermostatFanModes` attribute added.

Changes to the thermostatHeatingSetpoint Capability:

- `heatingSetpointMin` and `heatingSetpointMax` attributes replaced with `heatingSetpointRange`.

Changes to the thermostatMode Capability:

- `supportedThermostatModes` attribute added.

Changes to the thermostatSetpoint Capability:

- `thermostatSetpointMin` and `thermostatSetpointMax` attributes replaced with `thermostatSetpointRange`.

Changes to the thermostat Capability:

- `coolingSetpointMin` and `coolingSetpointMax` attributes replaced with `coolingSetpointRange`.
- `heatingSetpointMin` and `heatingSetpointMax` attributes replaced with `heatingSetpointRange`.
- `thermostatSetpointMin` and `thermostatSetpointMax` attributes replaced with `thermostatSetpointRange`.
- `supportedThermostatFanModes` attribute added.
- `supportedThermostatModes` attribute added.

[GitHub Release Tag](#)

June 08 2017

- *HubAction reference documentation* (page 1029) updated to clarify that `HOST` parameter is part of the headers map.

[GitHub Release Tag](#)

May 04 2017

- Asynchronous HTTP requests now support optional response handler methods. For cases when you just need to make a request, but don't care about the response, just pass `null` for the response handler. Docs updated [here](#) (page 987).
- Creating a Composite Device Handler? Check out the new *Composite Device Tiles* (page 589) documentation!
- **Some changes and additions to several Capabilities:**
 - `getAllActivities()` and `getCurrentActivity()` Commands removed from `mediaController` Capability.
 - `startActivity()` Command updated to accept the ID of the activity, instead of the name.
 - Optional `coolingSetpointMin` and `coolingSetpointMax` attributes added to the `thermostatCoolingSetpoint` Capability.
 - Optional `heatingSetpointMin` and `heatingSetpointMax` attributes added to the `thermostatHeatingSetpoint` Capability.
 - Optional `thermostatSetpointMin` and `thermostatSetpointMax` attributes added to the `thermostatSetpoint` Capability.
 - Optional `coolingSetpointMin`, `coolingSetpointMax`, `heatingSetpointMin`, `heatingSetpointMax`, `thermostatSetpointMin`, and `thermostatSetpointMax` attributes added to the `thermostat` Capability.

[GitHub Release Tag](#)

April 20 2017

- *Image capturing and viewing documentation* (page 575) is here! Learn how to store, retrieve, and display images from a LAN- or Cloud-connected camera device.
- Updated *Rate Limiting documentation* (page 609) with new child SmartApp and Device Handler limits, as well as clarify existing rate limits.

[GitHub Release Tag](#)

March 22 2017

- Composite Devices are here! Composite Devices allow developers to better model devices through a parent-child relationship between Device Handlers. Check out the *documentation* (page 583) and leverage this new design pattern for your composite devices!
- SmartThings has a new set of color standards for Device Handler Tiles. The *Color standards* (page 495) documentation covers all the new color standards.
- Updates to the *Writing Style Guide* (page 1063) and existing documentation to conform to new guidelines.

[GitHub Release Tag](#)

March 08 2017

- Do you have custom LAN device integrations? If so, check out the *Automatic LAN Device Discovery* (page 573) documentation to see what (if any) impact this has on your custom code.

[GitHub Release Tag](#)

March 02 2017

- Does your SmartApp or Device Handler need to execute every minute? Instead of writing your own cron expression, use the new *runEvery1Minute()* (page 929)!
- Need to convert color values between hexadecimal and RGB? The *ColorUtilities* (page 1001) class has what you need.
- If you are writing a parent-child SmartApp, check out the *expanded and clarified documentation* (page 394) for using the `app()` input type.
- A new capability, `bridge`, allows devices to declare they act as a bridge to other devices.
- A new attribute, `held`, has been added to the button capability!
- The *Writing Style Guide* (page 1063) has been updated with guidelines for document title and headings capitalization and formatting. If you are a contributor to these docs, make sure you check it out!

[GitHub Release Tag](#)

February 10 2017

- Did you notice? We've updated the *docs homepage* (page ??) to help readers quickly identify and navigate to common areas of interest.

[GitHub Release Tag](#)

February 08 2017

- Z-Wave fingerprinting updates! The *Z-Wave fingerprinting* (page 469) documentation has been expanded and updated with the latest information.
- Get information about a Device's status and last activity using the new *getStatus()* (page 1011) and *getLastActivity()* (page 1010) methods.
- New to Device Handler development, or looking for a refresher? We've overhauled our *Quick Start* (page 451) to ensure you can get up and running quickly and pain-free.
- Do you use cron to create recurring schedules? Have you seen if you could replace that often-difficult to understand, write, and maintain cron expression with any of our *runEvery** (page 348) methods? We've updated the *documentation* (page 347) to highlight these methods and encourage their use, instead of using cron.
- Did you know you can copy code examples right to your clipboard? We updated the UX to increase the visibility of this handy feature.

[GitHub Release Tag](#)

January 23 2017

- Search, discover and communicate with the devices in your network with the `HubAction` class. Check out the *new reference document for HubAction* (page 1029).
- If you need to get the account ID associated with an installed SmartApp, check out the `getAccountId()` (page 1031) method available on the *InstalledSmartApp* (page 1031) object!
- We've updated the *Editor and Simulator* (page 257) guide to clarify that you need to ensure you are on the correct shard when creating SmartApps or Device Handlers.
- A new Capability, `infraredLevel`, is now available!

[GitHub Release Tag](#)

January 03 2017

- Thinking about setting up a regular on and off schedule for your SmartThings? See our latest update, with examples, in *Schedule using cron* (page 349).
- Confused about sharding and where to publish your SmartApp or Device Handler? Here is a big picture view that clarifies *Publishing Custom Code* (page 623).
- Did you know there's a default delay between commands when you send a sequence of them to the Hub? See *sendHubCommand()* (page 936) reference documentation for details.

[GitHub Release Tag](#)

December 08 2016

- Quick, how do you know what Capabilities are supported by SmartThings? Checkout out the new generated *Capabilities Reference* (page 655), now live.
- Don't know much about ZigBee? We got you covered with our updated ZigBee documentation in the *ZigBee Primer* (page 523) and *ZigBee Reference* (page 1046) guides.
- What you, as a developer, must know while working with the SmartThings IDE. Checkout latest in the *Hubs and Locations* (page 247) guide.

[GitHub Release Tag](#)

November 30 2016

- Did you know you can refresh any page of the SmartApp on the mobile device with a set interval? See the *dynamicPage() options* (page 312) guide.

[GitHub Release Tag](#)

November 17 2016

- Changed code blocks to use the monokai dark theme.

[GitHub Release Tag](#)

November 15 2016

- Added ability to copy code blocks to the clipboard.

[GitHub Release Tag](#)

November 14 2016

GitHub Release Tag

- Added documentation for *working with time zones* (page 357).
 - Fixed warnings related to lexical parsing of code blocks.
-

November 10 2016

GitHub Release Tag

- Documented new *getModelName()* (page 1010) and *getManufacturerName()* (page 1010).
 - Styling and organiational changes to the left-hand navigation.
 - Internal build error fixes.
-

November 03 2016

GitHub Release Tag

- Revised `timeTodayAfter()` method description in the *SmartApp* (page 897) Guide
 - Added *Working With Time* (page 355) guide to the SmartApp Developers Guide
 - Fixed up scheduling reference docs in *Device Handler* (page 947), and *SmartApp* (page 897) Guides
 - Clarify getting latest device state in *Device* (page 1003), and *Working with Devices* (page 329)
 - Corrected `timeZone()` method description in the *SmartApp* (page 897) Guide
-

October 26 2016

GitHub Release Tag

- Documentation for *nextOccurrence()* (page 926).
 - Documentation for *getAllChildApps()* (page 904), *findAllChildAppsByName()* (page 902), *findAllChildAppsByNamespaceAndName()* (page 903), *findChildAppByNamespaceAndName()* (page 904), and *getAllChildApps()* (page 904).
 - Updated documentation for *getChildApps()* (page 904) to reflect that only “complete” child app installations will be returned.
 - Changed reference API docs to use getter forms instead of property access.
 - New attribute values added for the lock capability.
 - Typo fixes and other copy edits.
-

October 17 2016

GitHub Release Tag

- Documentation for *beta asynchronous HTTP APIs* (page 371)
 - Typo fixes and other copy edits
-

October 13 2016

GitHub Release Tag

- Moved rate limiting documentation into its own *guide* (page 609)
 - Typo fixes and other copy edits
-

October 11 2016

GitHub Release Tag

- Documented *SMS rate limits* (page 615)
 - Fixed typos
-

October 06 2016

GitHub Release Tag

- Added instructions for creating a simple code example when *creating a developer support ticket* (page 229).
 - Added *documentation* (page 308) for specifying a custom Remove button for preferences.
-

October 05 2016

GitHub Release Tag

- Added documentation for *passing data to schedule handler methods* (page 350).
 - Added *best practices* (page 651) for parent-child relationships.
 - Updated the repository's README with pull request guidelines.
 - Added scheduling APIs to the *Device Handler* (page 947) reference documentation (including all `runEvery*` APIs, which are now supported in Device Handlers).
 - Fixed broken cron tutorial link the *Scheduling* (page 345) guide.
 - Added note to the *first SmartApp tutorial* (page 175) and *Editor and Simulator* (page 257) that the Simulator is inconsistent with the mobile application.
-

September 23 2016

GitHub Release Tag

- Added link to the Z-Wave public spec on the following Z-Wave pages: *Building Z-Wave Device Handlers* (page 511) and *Z-Wave Primer* (page 507)
 - Updated the Color Control capability to correctly reflect the capability definition.
 - Updated Jinja template to add some more features for the ongoing generated capability documentation project.
 - Fixed minor grammatical errors.
-

September 14 2016

GitHub Release Tag

- Update to the *State and Atomic State documentation* (page 315) to reorganize, clarify, and expand content.
-

September 09 2016

GitHub Release Tag

- Removed Occupancy capability
- Fixed *unschedule()* (page 946) docs to clarify that a specific handler method name can be passed to `unschedule()`.

September 02 2016 (3)

GitHub Release Tag

- Fixing RTD build
-

September 02 2016 (2)

GitHub Release Tag

- Fixing RTD build
-

September 02 2016

GitHub Release Tag

- Typos and spelling fixes
 - Added more around the generated capabilities documentation framework
 - Added *Troubleshooting* (page 445) document to the SmartApp Web Services guide
 - Fixed colorControl example code in the capabilities reference
-

August 17 2016

GitHub Release Tag

- Fix *documentation* (page 943) for `subscribeToCommand()` (only takes a `Device` argument, not a list of `Devices`)
 - Typos and spelling fixes
-

August 16 2016

GitHub Release Tag

- *Documentation* (page 261) for the ability to pass a `Throwable` to logging methods to get more logging details about the exception shown in the logs.
-

August 15 2016

GitHub Release Tag

- Make edits to Makefile as a first step in getting generated capabilities documentation integrated into the documentation build.
-

August 04 2016

GitHub Release Tag

- Added *zigbee.parseZoneStatus()* (page 1053) documentation
 - Added documentation for *Additional ZigBee classes* (page 1053)
 - Clarified *findChildAppByName()* (page 903) API documentation
 - Added *documentation* (page 535) to Device Handler Guide for other useful APIs available to Device Handlers, including Scheduling, HTTP Requests, and State.
 - Fixed documentation for *Event.dateValue* (page 1018) to indicate that it returns `null` if date cannot be parsed
 - Various fixes for `reStructuredText` formatting and legal syntax warnings
 - Moved this documentation change log to top of navigation
-

July 28 2016

GitHub Release Tag

- Document the new *hideWhenEmpty* (page 307) preferences option.
-

July 25 2016

GitHub Release Tag

- Add a strong warning to the *State documentation* (page 315) to emphasize the importance of never mixing `atomicState` and `state` in the same `SmartApp`.
-

July 21 2016

GitHub Release Tag

- *Documented* (page 433) the new redirect URI field on OAuth SmartApps
-

July 07 2016

GitHub Release Tag

- Added documentation for working with collections in *State* (page 322) and *Atomic State* (page 323).
 - Added documentation for *AppState* (page 981)
 - Added documentation for *InstalledSmartApp* (page 1031)
 - Added *clarification* (page 422) that the callable URL for Web Services SmartApps will vary by installed location
 - Updated developer call schedule
-

June 23 2016

GitHub Release Tag

- **Splitting the Music Player capability into three capabilities**
 - Audio Notification
 - Music Player
 - Tracking Music Player
-

June 17 2016

GitHub Release Tag

- Adding WOL (Wake On Lan) documentation
-

June 13 2016

GitHub Release Tag

- Adding *Code Review Guidelines and Best Practices* (page 631) for SmartApps and Device Handlers.
-

June 9 2016

GitHub Release Tag

- Fix spelling of “capability” in *Attribute* (page 997) docs
 - Fix capitalization of “localIP” in *Hub* (page 1027) docs
 - Document the *SmartThings developer support* (page 229) form
 - Document *Device Handler Preferences* (page 499)
 - Document *device-specific preference inputs* (page 306)
 - Clarify *GitHub Integration* (page 265) only available in the US
-

May 27 2016

- Add `additionalParams` argument for ZigBee library. [Docs](#) (page 1046) | [GitHub PR](#)
-

May 23 2016

- Updated and expanded Device Handler tiles docs. *Docs* (page 473) | [GitHub PR](#).

Part II

Overview

SmartThings is the open developer platform for the Internet of Things.

With SmartThings, developers can:

- Create applications that let users connect devices, actions, and external services to create automations.
 - Integrate new devices into the SmartThings ecosystem.
 - Publish applications and device integrations to the SmartThings catalog.
-

Developer highlights

SmartThings was built to be developer-friendly. Some of the key developer features:

- A simple programming framework using the Groovy programming language. Don't know Groovy? No worries. We've written a *tutorial* (page 115) to get you up to speed.
 - An architecture that allows developers to control hardware with simple software. Turning a switch on is as easy as `switch.on()`.
 - A web-based IDE for developing SmartThings solutions.
 - A Simulator for testing your code, *even if you don't have specific devices you are developing for*.
 - An active and growing [community](#) of SmartThings developers.
-

How it works

There are two primary ways that developers can create with SmartThings.

46.1 SmartApps

```
def someoneArrived(evt) {  
    lights.on()  
    sendPush("Someone has arrived!")  
}
```

SmartApps are small programs that allow users to connect their devices to make their home more intelligent. As the world around us becomes more and more connected, it is the intelligence *between* these devices that makes our world smart. SmartApps allow developers to control hardware with simple software.

SmartApps can typically be summarized by what they do. Some example SmartApps:

- “Turn the lights off after a certain time when no motion is detected”
- “Notify me if a door opens when I’m not home”
- “Turn my thermostat down when I leave home”

SmartThings ships with many SmartApps already available. Almost all automations that you configure with your SmartThings mobile application are SmartApps. If you’ve set up your lights to come on when motion is detected, or to receive a notification if your door opens when you aren’t home, you’ve used SmartApps.

Of course, SmartApps are capable of much more than the above examples. SmartApps can communicate with external web services, send push and SMS notifications, expose their own REST endpoints, and more.

46.2 Device Handlers

```
def on() {  
    zigbee.on()  
}
```

Developers can also integrate new devices into the SmartThings ecosystem by creating *Device Handlers*. These Groovy programs encapsulate the details of communication between SmartThings and the physical devices. In the SmartApp code example above, we turned the lights on by simply calling `lights.on()`. The Device Handler is responsible for physically turning the light on (don’t worry about the details of this just yet).

An open platform

SmartThings was built by developers, for developers. We recognized that only by creating an open development platform, will the power of the IoT be fully unleashed.

Our *web-based IDE and simulator* (page 251) allows developers to create, edit, test, and publish their SmartThings code. SmartApps and Device Handlers are hosted in our [public GitHub Repository](#), and our web-based IDE and Simulator is *integrated with GitHub* (page 265).

Our vibrant [developer community](#) is a great place to learn, collaborate, and help each other.

What's next

To start developing with SmartThings, you will need to create a developer account and become familiar with the developer tools. This is covered next in the *Up and Running* (page 105).

SmartThings uses the Groovy programming language. Don't know Groovy? Check out our *Groovy Basics* (page 115) and *Groovy With SmartThings* (page 161) tutorials.

Then, take a deep dive into developing with SmartThings by writing your first SmartApp, using the *Writing Your First SmartApp* (page 175).

Part III

Up and Running

SmartThings offers a rich toolset to develop, test, and publish custom code.

Don't have a SmartThings Hub or any devices yet? Carry on! You can still create an account and even develop without any hardware, by using our online IDE and Simulator.

Of course, you'll want to have the hardware sooner than later, but you can start developing with SmartThings with nothing more than the free SmartThings mobile app, a web browser, and an internet connection.

Register

If you already have the SmartThings mobile app, you can access the developer IDE at <https://graph.api.smartthings.com>, using the same email and password.

If you don't have the mobile app, you can register for an account by visiting <https://account.samsung.com/account/signUp.do>. You can then download the free SmartThings mobile app for iOS, Android, or Windows.

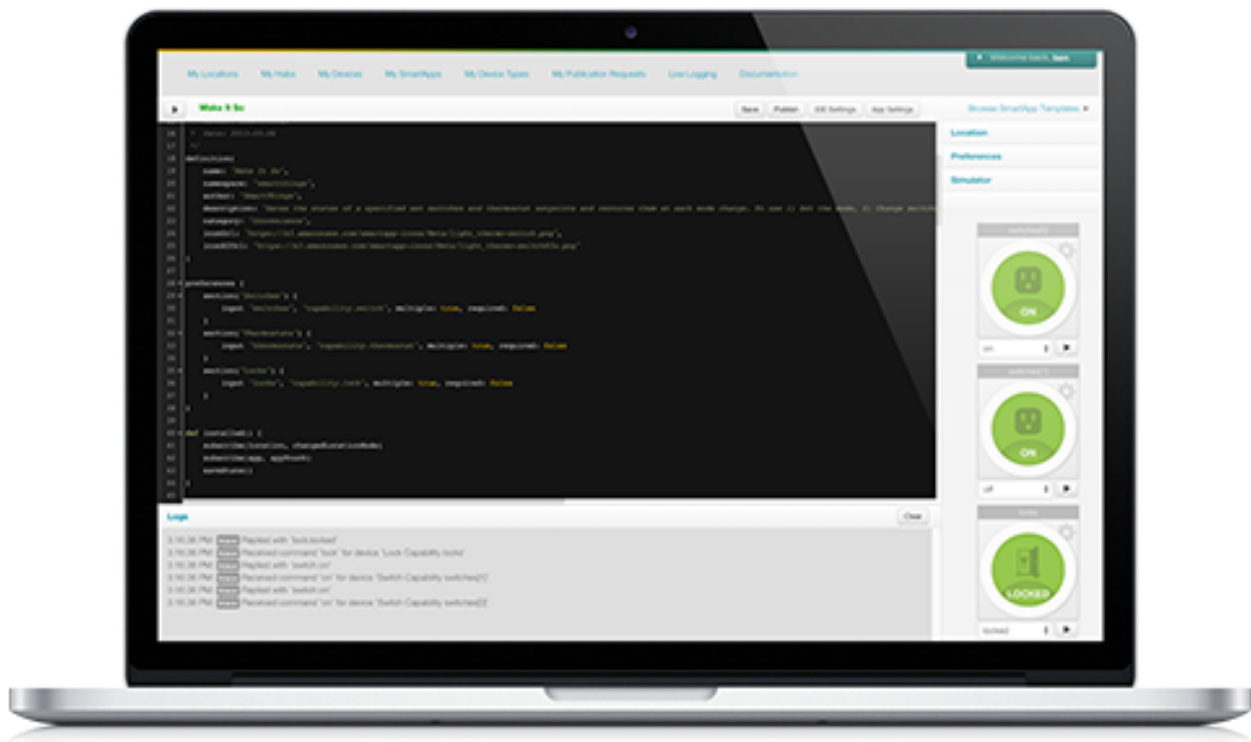
Explore

The *Tools and IDE* (page 251) guide discusses the developer tools in more detail, but for now, let's look at a few key features to get you comfortable.

50.1 Account management

You can use the tools available to view and manage the Locations, Hubs, and Devices, as well as view a live log for locations where you are listed as the owner.

50.2 IDE and Simulator



At the top of the page, you'll notice links for *My SmartApps* and *My Device Handlers*. This is where any custom code will be listed. Clicking on any SmartApp or Device Handler will bring you to the code editor, where you can view, edit, test, and publish your custom code.

As a new SmartThings developer, you won't have any SmartApps or Device Handlers yet. We will guide you through creating one later in the [Writing Your First SmartApp](#) (page 175).

Next steps

Now that you know what the SmartThings developer platform offers, you can dive in to the fun stuff.

If you're new to Groovy, we recommend that you read through the *Groovy Basics* (page 115) tutorial. You'll learn about Groovy, and how SmartThings uses it for development. The *Groovy With SmartThings* (page 161) tutorial discusses some key differences between regular Groovy and Groovy with SmartThings.

Once you've completed that (or maybe you're the adventurous sort and just want to dive right in to some SmartApp code), check out the *Writing Your First SmartApp* (page 175) tutorial.

Part IV

Groovy Basics

SmartThings uses the Groovy programming language. If you've programmed before, you can learn Groovy.

The Groovy programming language is documented at <http://www.groovy-lang.org/documentation.html>. This tutorial will familiarize you with Groovy and its use in SmartThings, but is not a complete reference for the language.

Tip: If you already know Groovy, or prefer to learn as you go, you can skip this tutorial and refer to this page as a mini-reference of sorts. It is important, however, that you understand how Groovy is used in SmartThings. That is discussed in the *Groovy With SmartThings* (page 161) tutorial.

To develop with SmartThings, you do not need to be an expert in Groovy. The SmartThings development environment was created to be easy-to-use, so that it does not require someone to be proficient in Groovy (or any other language). That said, having a basic understanding of some of the core concepts of Groovy will help you be most productive in your development.

Overview

Groovy is an object-oriented programming language for the Java platform. It is a dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk.

If you are familiar with languages like Java, C/C++, Python, Ruby, or JavaScript, you will see many similarities in Groovy.

Groovy code is compiled to byte code that is executed by the Java Virtual Machine (JVM). We choose Groovy as the SmartThings programming language for its simplicity and flexibility, as well as the performance and stability of the JVM.

Because Groovy is compiled to byte code that runs on the JVM Java Virtual Machine (JVM), 99% of Java code is valid Groovy. The standard Java libraries are available to Groovy programs. Groovy extends Java in many useful ways, which we'll learn about here.

Installing Groovy

The best way to get familiar with Groovy is by installing it and experimenting. SmartThings development does not require you to have a copy of Groovy installed, since SmartThings code is executed within SmartThings infrastructure, but having a local copy of Groovy is useful for learning.

Head over to the [Groovy Documentation](#) site and follow the Getting Started guides for downloading and installing Groovy (the rest of the Getting Started material is pretty awesome too, and definitely worth a read).

We make heavy use of the Groovy Console to test things out, and recommend you do to.

Note: In the code snippets below, you'll see a method `assert()` used often. This method is built in to Groovy, and we use it to verify assumptions. If the value passed to `assert()` is not true, the program will terminate. This lets us test out our code easily.

For example, `assert true` is valid, and the program will continue. Anything that evaluates to false will cause the program to halt, so `assert false` will terminate with an informative message.

While useful for learning, it's important to note that `assert()` is **not available** for you to use in SmartThings code. Neither is the method `println()`, for that matter. For security and performance reasons, SmartThings runs in a sandboxed environment that restricts access to certain features. The sandboxed environment is discussed further in the *Groovy With SmartThings* (page 161) tutorial.

Optional semicolons

Semicolons are optional in Groovy, and generally not used:

```
def someString = "this statement has a semicolon";  
def someOtherString = "this one does not"
```

Comments

Groovy supports single line comments:

```
// this is a single line comment
// each line requires slashes
def myNum = 2 // comments can also come at the end of a statement
```

Multiline comments are also supported:

```
/* this is a comment that
   spans multiple lines.*/
def myNum = 2
```

Objects

In Groovy, everything is an object. Objects have *methods* and *properties*.

Methods are the things the object can do, and similar to other languages, are optionally (more on that later) invoked with parentheses () that may contain arguments.

```
// calling method doSomething on someObject
someObject.doSomething()

// calling method doSomethingElse with one argument
someObject.doSomethingElse("a string argument")

// get the property named someProperty on someObject
someObject.someProperty
```

Optionally typed

Groovy is an **optionally typed** language. The following are both valid Groovy:

```
// explicit typing
Person person = new Person ()

// using def
def person2 = new Person ()
```

In Groovy, we can use `def` in place of an explicit type. The exact type of object that will be assigned will vary when using `def`.

Why use `def` instead of explicit types? While not required, `def` is commonly used in Groovy (and in SmartThings) because it provides greater flexibility and readability.

Consider this strongly typed example:

```
String addThem(String str1, String str2) {
    return str1 + str2
}

String added = addThem("Smart", "Things");
assert "SmartThings" == added
```

In the example above, `addThem()` is defined to accept two `String` parameters. Groovy supports operator overloading, so using the `+` operator concatenates the two strings.

What happens when we try to invoke `addThem()` with two numbers?

```
// fails!
assert 3 == addThem(1, 2)
```

This results in an exception like this:

```
groovy.lang.MissingMethodException: No signature of method: Script1.addThem() is applicable for argument
Possible solutions: addThem(java.lang.String, java.lang.String)
at Script1.run(Script1.groovy:7)
```

Because `addThem()` is defined to accept two `String` parameters, we get a `MissingMethodException` when calling `addThem(1, 2)`, since there is no method named `addThem` that accepts two numbers.

If we use `def` instead of an explicit type, we can take advantage of something called **duck typing**. Put simply, duck typing is the principle that if it walks like a duck and quacks like a duck, then it's a duck. In programming terms, this means that if an object supports certain properties or methods, then we can use those regardless of its type.

To illustrate this with an example, consider the above example refactored to use `def`:

```
def addThem(str1, str2) {
    // strings and numbers support the + operator
    return str1 + str2
}

def added = addThem("Smart", "Things")
assert added == "SmartThings"

def added2 = addThem(4, 2)
assert added2 == 6
```

Omitting the explicit type information in favor of `def` allows us to build flexible programs without getting bogged down in ensuring we have all our typing information correct. This is particularly useful for smaller programs, which is what you will be writing with SmartThings.

Note: Strict statically typed languages like Java determine the method that will be called at *compile time*. Groovy determines the methods to invoke at *runtime*, using something called multi-methods or dynamic dispatch. You can read more about multi-methods [here](#) in the Groovy documentation.

Operators

Groovy supports all the typical operators, such as arithmetic operators, assignment operators, and relational operators:

```
assert 1 + 2 == 3 // use == for checking equality
assert 1 < 2

def a = 1
def b = a += 2
assert a == 3

def c = 4
def d = c++
assert d == 5
```

There are a few other notable operators that you may not have seen in other languages; one of them is the Safe Navigation Operator. Using Groovy's Safe Navigation Operator, you can navigate object structures without fear of getting a `NullPointerException` on a null object.

Suppose we have a property named `location`, that also has a method `getHelloHome()`. Further, suppose that the object returned by `getHelloHome()` has a method named `getPhrases()`. Ultimately, we want to get the phrases.

We could do:

```
def phrases = location.getHelloHome().getPhrases()
```

But, what if `getHelloHome()` returns null? We'd then get a `NullPointerException` at runtime when trying to call `getPhrases()` on a null object.

If you're not familiar with Groovy, you might try something like this to avoid that:

```
def hh = location.getHelloHome()
def phrases
// recall that non-null objects are "true"
if (hh) {
    phrases = hh.getPhrases()
}
```

That works, and is valid Groovy, but we can do better. Using the safe navigation operator (`?.`), we can safely traverse the object graph. If any objects are null, the method simply will not be invoked and `null` will be returned.

This results in much cleaner code:

```
def phrases = location.getHelloHome()?.getPhrases()
```

In this example, if `getHelloHome ()` is not null, we'll call the `getPhrases ()` method on it. If it does return null, the whole expression simply returns `null`.

If there's ever a chance of running into a `NullPointerException` when navigating an object structure, use the safe navigation operator to safely (and concisely) avoid it.

There are many more Groovy operators documented [here](#).

Strings

Strings can be defined using single, double, or triple quotes:

```
def a = "some string"
def b = 'another string'
def c = '''Triple quotes
        allow multiple
        lines'''
```

Strings defined with double quotes support interpolation. This allows us to substitute any Groovy expression into a String at the specified location. Interpolation is achieved using the `${}` syntax:

```
def name = "Your Name"
def greeting = "Hello, ${name}"
assert "Hello, Your Name" == greeting
```

Of course, more interesting interpolations are possible. Any expression can be placed inside the `${}`:

```
def name = "Your Name"
def greeting = "Hello, ${name.toUpperCase()}"
assert "Hello, YOUR NAME" == greeting
```

You can also use the `$` without the `{}` for simple property substitutions or simple dotted expressions:

```
def name = "Your Name"

// can omit the {} here
def greeting = "Hello, $name"
assert "Hello, Your Name" == greeting

def person = [firstName: 'Walter', lastName: 'Sobchak']
def greeting = "Hello, $person.firstName $person.lastName"
```

Note: Dotted expressions are expressions of the form `a.b` or `a.b.c`. Expressions that would contain parentheses like method calls, curly braces for closures, or arithmetic operators, are not dotted expressions and you should use `${}`. We recommend always using the `${}` notation.

You'll see String interpolations frequently in SmartThings.

There are some other handy Groovy String features, like the ability to remove part of a string using the `-` operator:

```
def lannisters = "A Lannister does not always pays their debts"
def corrected = lannisters - "does not "
assert "A Lannister always pays their debts" == corrected
```

You can read more about Strings [here](#).

Lists and Maps

Groovy supports the typical collection structures like Lists and Maps in an easy-to-use way.

Here are some examples showing how to work with Lists in Groovy:

```
// simple list of Numbers
def myList = [2, 3, 5, 8, 13, 21]

// use the << operator to append items to a list
myList << 34
assert myList == [2, 3, 5, 8, 13, 21, 34]

// get elements in a list
// first element is at index 0
assert 8 == myList[3]

// can use negative index to start from the end
assert 21 == myList[-2]

// lists can support different types of data
def myMixedList = [1, "two", true]
```

Maps are similarly straightforward:

```
// simple map of key/value pairs
def myMap = [key1: "value1", key2: "value2"]

// can get value for a key with the "." notation:
assert "value1" == myMap.key1

// can also get the value using subscript notation:
assert "value2" == myMap['key2']

// a list of maps
def listOfMaps = [[key1: "val1", key2: "val2"],
                  [key1: "another val", key2: "and another"]]
assert "another val" == listOfMaps[1].key1
```

While lists and maps are simple in Groovy, there are many powerful methods in the Groovy collections APIs that extend their power. You are encouraged to read the Groovy documentation for more information, but here are some cool examples:

```
def colors = ["red", "green", 42, "blue"]

// remove items from a list with the "-" operator
```

```
colors = colors - 42
assert ["red", "green", "blue"] == colors

def people = [[first: "Jimmy", last: "James"],
              [first: "Bill", last: "McNeal"]]

// The * operator allows us to invoke an action on every item in the
// collection, returning a new list of results.
def firstNames = people*.first
assert ["Jimmy", "Bill"] == firstNames

// this is also useful for invoking the same method on a collection of objects:
def listOfStrings = ["a", "b", "c"]
assert ["A", "B", "C"] == listOfStrings*.toUpperCase()
```

Control structures

Groovy supports the conditional if/else syntax as you'd expect:

```
if (...) {
    ...
} else if (...) {
    ...
} else {
    ...
}
```

You can also use the `switch` statement to handle possible values conditionally:

```
def deviceDescription = "presence: 1"
def result = ""

switch (deviceDescription) {
    case "presence: 0":
        result = "not present"
        break
    case "presence: 1":
        result = "present"
        break
    default:
        result = "unknown"
}

assert "present" == result
```

Looping is also similar to Java or C:

```
def result = ""
for (int i = 0; i < 3; i++) {
    result += "Z"
}
assert "ZZZ" == result
```

You can also use the `for/in` loop when working with collections:

```
def next = 0
for (i in [8, 13]) {
    next += i
}
assert next == 21
```

Calling methods

When invoking methods, parentheses are *sometimes* optional. Methods that do not accept any parameters must include the parentheses.

```
def myMethod() {  
    // ...  
}  
  
def myOtherMethod(someArg1, someArg2) {  
    // ...  
}  
  
myMethod()           // OK  
myMethod             // error  
myOtherMethod(2, 3) // OK  
myOtherMethod 4, 5  // OK
```

Getters and setters

Groovy adds in some convenience JavaBean style getter and setter methods. It's worth being aware of this in case you see some code that references a property that seemingly isn't defined anywhere:

```
def getSomeValue() {  
    return "got it"  
}  
  
assert "got it" == someValue
```

How did referencing `someValue` end up invoking the method `getSomeValue()`? When Groovy sees a reference to the property named `someValue`, it first looks to see if it is defined somewhere. In the above example, it is not. So, Groovy then looks to see if there is a getter method. JavaBean conventions specify that a properties getter method should be named beginning with “get”, followed by the name of the property (with the first letter of the property capitalized).

Don't worry if that's somewhat confusing; just know that if you a reference to a property name that doesn't appear to exist, it might be invoking a getter method.

Defining methods

Methods are generally defined and invoked as in other modern languages, with some notable enhancements.

First, the basics. Method signatures can accept both typed and untyped arguments:

```
// arguments types are optional:
def asMap(arg1, arg2) {
    return [arg1: arg2]
}
assert [key: "val"] == asMap("key", "val")

// can use typed arguments as well
Map asMapWithTypedArgs(String arg1, String arg2) {
    return [arg1: arg2]
}
assert [key: "another val"] = asMap("key", "another val")
```

The return statement is optional in a Groovy method. The value of the last expression evaluated is returned by default:

```
def asMap(arg1, arg2) {
    // no return statement
    [arg1: arg2]
}
assert [key: "val"] == asMap("key", "val")
```

Methods can also be defined to accept *named parameters*. This is frequently used in SmartThings, as it allows for flexible and easily-extendable methods. This is accomplished by accepting a Map parameter (the typing is optional, but used here for clarity):

```
def myMethod(Map params) {
    "$params.firstName, $params.lastName"
}

// note the lack of parentheses here also
assert "First, Last" == myMethod firstName: "First", lastName: "Last"
```

Methods can also define default values for parameters. If not passed when calling the method, the default will be used:

```
def defaultParams(first, last, middle = "") {
    "Welcome, $first $middle $last"
}

def greetGeorge = defaultParams("George", "Costanza", "Louis")
def greetKramer = defaultParams("Cosmo", "Kramer")
```

```
assert "Welcome, George Louis Costanza" == greetGeorge
assert "Welcome, Cosmo Kramer" == greetKramer
```

Worth noting is that none of the above definitions include any type of explicit visibility modifier information. By default, when using `def`, the method is public. Want to make your method private? It's syntactically allowed, but actually isn't respected by Groovy (gasp!). And in SmartThings, this really isn't necessary since we are not creating our own classes or object models. So, we typically just omit any visibility modifier for simplicity.

Exception handling

Like other programming languages, Groovy has error conditions, or exceptions. Because Groovy is based on Java, there are similarities to how Java handles exceptions. The big difference is that Groovy *does not require you to handle so-called checked exceptions*. In Groovy, we are always free to handle exceptions if we want, or disregard them and let them percolate up the call stack.

To handle general exceptions, you can place the potentially exception-causing code in a try/catch block:

```
try {
    someMethodThatMightGoBoom()
} catch (=)
    // log the error message, and/or handle in some way
}
```

By not declaring the type of exception we can catch, any exception will be caught here.

Closures

If you are most familiar with languages like C or Java, closures may be something you haven't heard of or used. You'll see a *lot* of closures being used in Groovy and SmartThings, so it's worth understanding the basics.

First, consider a simple example. Say we have a List of numbers, and want to do something with each item in the list. For our purposes, it doesn't matter what we want to do, only that we want to iterate over every item in the list and do something.

We could certainly do something like this:

```
def list = [1, 2, 3, 4]
for (int i = 0; i < list.size(); i++) {
    println list[i]
}
```

That works, but if you think about it, our code shouldn't have to know the details of the list's size or control iterating over its contents. All we really care about is doing something to each item!

Fortunately, because Groovy supports closures, we can rewrite the above code as:

```
def list = [1, 2, 3, 4]
list.each { num ->
    println num
}
```

If you have a Java background, you might be thinking to yourself that Java already solves this with the `for/each` statement. And for simple iteration, you're right - both the `for/each` statement in Java and the `each()` method in Groovy appear to do the same thing. But, closures are much more powerful than just providing more convenient ways to iterate, as we'll see next.

Consider an example where given a list of numbers, we want to know which numbers are greater than 50. Without closures, we would probably write something like this:

```
def greaterThan50(nums) {
    def result = []
    for (num in nums) {
        if (num > 50) {
            result << num
        }
    }
    result
}

def test = greaterThan50([2, 5, 62, 50, 25, 88])
assert 2 == test.size()
```

```
assert test.contains(62)
assert test.contains(88)
```

This is valid Groovy, but with the ability to use closures, we can write code that is much more expressive and concise:

```
def greaterThan50(nums) {
    // findAll returns a list of items
    // that match the condition specified in the passed-in closure
    nums.findAll {
        it > 50
    }
}

def test = greaterThan50([2, 5, 62, 50, 25, 88])
assert 2 == test.size()
assert test.contains(62)
assert test.contains(88)
```

This may look very foreign to you, but once you start using and understanding closures, you'll find them *very* useful.

Simply put, Groovy Closures are anonymous blocks of code that can be passed to other methods, and those methods can then call that block of code.

The example above uses the `findAll()` method that is available on all Groovy collections. The method accepts a closure (defined within `{}`) as the argument (when passing closures to methods, it is typical and preferred to *not* put parentheses around the parameters).

`findAll()` works by calling the passed-in closure on every element in the list, and if the item meets the criteria specified in the closure (greater than 50), adds it to a new list that is returned. The closure (`{ it > 50 }`) is passed the item - by default, this is available in a variable named `it`. You can also provide a name if you wish, by using the `->` operator:

```
nums.findAll { num ->
    num > 50
}
```

To deepen our understanding, we will next look at an example of creating a method that accepts a closure.

Let's say we want to print all even numbers up to a specified number¹. While we can do this without closures, using them will illustrate how they work.

Here's the code to do this:

```
def pickEven(n, block) {
    for (int i=2; i <= n; i += 2) {
        block(i)
    }
}

pickEven(10) {
    println i
}
```

The `pickEven()` method accepts an upper bound (`n`), and a closure (`block`). It iterates over all the even numbers up to the upper bound, and calls the passed-in closure on each (`block(i)`).

When we call `pickEven()`, the closure simply calls `println()` on each item. Running this would result in the following output:

¹ This example is taken from the book *Programming Groovy: Dynamic Productivity for the Java Developer* by Venkat Subramaniam.

```
2
4
6
8
10
```

A final note about closures, with regards to the use of the optional parentheses. As discussed earlier, parentheses are optional when calling methods in most cases. This is no different for closures, but convention is to *not* put parentheses around closures as arguments to methods.

The above call to `findAll()` could be written as:

```
nums.findAll({ num ->
    num > 50
})
```

It is idiomatic Groovy to not surround closure arguments with parentheses. When a method accepts multiple parameters, and the closure is the last parameter, the closure should be outside the parentheses.

```
// instead of:
pickEven(10 {
    println it
})

// prefer:
pickEven(10) {
    println it
}
```

There's much more to know about closures if you're curious, but if you understand the above concepts you will know enough to use them in your SmartThings development.

Groovy truth

Groovy has some special definitions for what is true and what is false. It's worth understanding these definitions, as they become very valuable in writing concise, expressive Groovy code.

Boolean values behave as you'd expect:

```
def t = true
def f = false

assert t
assert !f
```

If an object reference is null, it will evaluate to false:

```
def obj
assert !obj
```

This allows us to remove some boilerplate code around null checks. If you're familiar with Java, you have probably seen code like this:

```
if (obj != null) {
    // ...
}
```

In Groovy, we can simply do:

```
if (obj) {
    // ...
}
```

Strings also provide some handy truthiness:

```
def str1 = ""
def str2 = "some string"

assert !str1 // empty strings are false
assert str2
```

Collections also support reasonable boolean values - empty collections evaluate to false:

```
def list1 = [1, 2, 3]
def list2 = []
def map1 = ['myKey': 'myValue']
def map2 = [:]

assert list1
```

```
assert !list2 // empty list is false
assert map1
assert !map2 // empty map is false
```

Back to Java, you may be familiar with writing code like this:

```
Map<String, String> myMap = someMethodThatReturnsAMap ();
if (myMap != null && !myMap.isEmpty()) {
    // ...
}
```

That's a lot of noise in the code just to check that the map is not empty. With Groovy, this becomes much more straightforward:

```
def myMap = someMethodThatReturnsAMap ()
if (myMap) {
    // here we know that the map is not null, and contains items.
}
```

The above should get you through 99% of the code you'll see and write with SmartThings, but see the Groovy documentation for [more on the Groovy Truth](#).

Default imports

Groovy imports several Java and Groovy packages by default. The following packages are imported for us (no need to explicitly import them via the `import` statement):

- `java.io.*`
 - `java.lang.*`
 - `java.math.BigDecimal`
 - `java.math.BigInteger`
 - `java.net.*`
 - `java.util.*`
 - `groovy.lang.*`
 - `groovy.util.*`
-

What about classes?

At the beginning of this tutorial, we said that Groovy is an object-oriented language. Yet, we haven't discussed creating classes in this tutorial. The reason for this is that in SmartThings, creating your own classes actually isn't possible. In SmartThings, each SmartApp or Device Handler is a relatively small, contained piece of code that runs in a sandboxed environment.

If you want to learn more about classes in Groovy in general or for usage outside of SmartThings, see the [Groovy documentation](#).

Further reading

There are many resources available to learn more about Groovy. As we'll see in the *Groovy With SmartThings* (page 161) tutorial, there are some things about the Groovy programming language that we simplify with SmartThings, so a full knowledge of Groovy and all its capabilities is not necessary to develop with SmartThings.

If you want to learn more about Groovy, here are some good resources available online:

- The [Groovy Documentation](#) is the official language documentation.
- The [Style Guide](#) in the Groovy documentation contains many useful guidelines and recommendations for writing idiomatic Groovy code.
- [Learn Groovy in Y minutes](#) is an excellent, concise, and code-heavy tutorial for getting familiar with Groovy.
- [Groovy for Java Developers](#) aims to get Java developers familiar with Groovy quickly.

There are also several books on Groovy. Here are a couple we know and recommend:

- [Groovy in Action](#)
 - [Programming Groovy](#)
-

Next steps

Now that you know some of the basics of Groovy, head over to our *Groovy With SmartThings* (page 161) tutorial to learn how SmartThings uses Groovy in some very specific ways for development.

Part V

Groovy With SmartThings

SmartThings runs Groovy in a sandboxed environment. This means that not all features of the Groovy programming language are available in SmartThings. To understand why, we need to understand where SmartThings code is executed.

All SmartThings code is executed in, and by, the SmartThings ecosystem. When you write a SmartApp or a Device Handler, it will ultimately be executed by the SmartThings platform. It may execute on the Hub or in the SmartThings cloud, but the important thing to note is that it is executed by SmartThings.

Because SmartApps and Device Handlers execute within the SmartThings ecosystem, SmartThings restricts access to certain methods or features. You can't create or open a file, for example.

Before we discuss the specifics of what is and what is not available to your SmartApps and Device Handlers, we'll first discuss how SmartThings makes several APIs available within your SmartApp or Device Handler. While this is not strictly necessary to understand to be able to develop with SmartThings, it may help to shed light on what is happening behind the scenes.

How it works

One of the first things you'll notice when starting to develop with SmartThings, is that there are many methods available to you that do not require any import statements. In fact, it's rare to see import statements at all in SmartThings.

This is because every SmartApp or Device Handler is actually an instance of an abstract *Executor* class defined in the SmartThings platform. This *Executor* class defines or includes many methods. The result of this is that every SmartApp or Device Handler has available to it (through inheritance) a large number of methods without importing anything.

This model provides a simple framework in which you can develop your SmartApps and Device Handlers - all the necessary methods are simply available to call without needing to import anything.

Now that we understand (at least at a high level) how SmartApps and Device Handlers make various methods available, let's look at some of the things that are *not* allowed within SmartThings code. After that, we'll look at the entire whitelist of allowable classes.

Language simplifications

73.1 Classes and JARs

As a SmartApp or Device Handler author, you cannot create your own classes, or import any custom JARs. While at first this may seem like a significant restriction, in practice you'll rarely find this to be the case. Because of the nature of SmartApps and Device Handlers, and the various methods available to you, the need to create your own classes or object structures is rarely needed.

There may be certain scenarios in which you discover your task would be easier if only you could import some third-party library or create your own helper class. In cases like these, reach out to us on the forums and let us know your specific use case. It's possible there already exists an API to do what you need, and if not, we may be able to get it added to SmartThings.

73.2 Restricted methods

Because SmartThings code executes within its own ecosystem, there are a few methods that we restrict for security purposes. Many of these methods deal with Groovy's advanced metaprogramming concepts. Groovy metaprogramming allows developers to get and modify runtime information for objects. In SmartThings, this isn't necessary to do and is a potential security risk, so they are disabled.

Here are the methods that are not available in SmartThings. Trying to access these will result in a `SecurityException` at runtime.

- `addShutdownHook()`
- `execute()`
- `getClass()`
- `getMetaClass()`
- `setMetaClass()`
- `propertyMissing()`
- `methodMissing()`
- `invokeMethod()`
- `mixin()`
- `print()`
- `printf()`

- `println()`
- `sleep()`

73.3 Global variables

73.3.1 Constants

Due to the sandboxed nature of SmartApp and Device Handler execution, defining global constant variables like this will **not** work:

```
def MY_CONSTANT = "some constant value"
```

Defining constants as above is valid Groovy code and will compile, but the value of `MY_CONSTANT` will be `null`.

Instead, for any global constants you'd like in your SmartApp or Device Handler, define a no-op getter method that returns the value:

```
def getMyConstant() {  
    return "some constant value"  
}
```

You can then call the method directly, or use some *Groovy magic* (page 139) to invoke no-arg getters.

73.3.2 Mutable variables

Similarly, creating a global variable and then updating it will **not** work:

```
def globalVar = "some value"  
  
def someMethod() {  
    // update the variable here, but this will not persist across executions!  
    globalVar = "some updated val"  
}
```

Instead, any information you need persisted between executions needs to be stored the application *state* (page 315).

73.4 Other notable restrictions

There are a few other notable restrictions in SmartThings worth discussing:

- You cannot create your own threads.
 - You cannot use `System` methods, like `System.out()`
 - You cannot create or access files.
 - You cannot define closures outside of a method. Something like `def squareItClosure = {it * it}` is not allowed at the top-level, outside of a method body.
-

Allowed classes

SmartThings also specifies a *whitelist* of allowed classes. Only classes included in this whitelist are available for use within SmartThings. Whenever a method is called (any method), SmartThings first checks to see that the *receiver* of the method (the object the method is being called on) is in the allowable types whitelist. If it isn't, a `SecurityException` will be thrown. This same principle applies to the creation of new objects with the `new` keyword - if the object being created is not in the whitelist, a `SecurityException` is also thrown.

Most SmartThings solutions will not need to instantiate any of these classes directly. The majority of objects you work with will be available to you via callback parameters or injected right into your SmartApp or Device Handler. Here is the whitelist of available, non-SmartThings-specific types (i.e., Java, Groovy and third party library classes):

Important: Certain methods that update JVM settings are disallowed, even though the usage of the class is permitted. For example, calling `TimeZone.setDefault()` is not allowed, and will throw a `SecurityException`.

This is due to the fact that many SmartThings applications may be executing on a single JVM. Updating system-wide properties may have unintended consequences on other applications running on the same JVM.

As a general rule-of-thumb, if a method has impact on the underlying JVM, it will not be allowed, for the reasons discussed above.

- `ArrayList`
- `BigDecimal`
- `BigInteger`
- `Boolean`
- `Byte`
- `ByteArrayInputStream`
- `ByteArrayOutputStream`
- `Calendar`
- `Closure`
- `Collection`
- `Collections`
- `Date`
- `DecimalFormat`
- `Double`

- Float
- GregorianCalendar
- HashMap
- HashMap.Entry
- HashMap.KeyIterator
- HashMap.KeySet
- HashMap.Values
- HashSet
- Integer
- JsonBuilder
- LinkedHashMap
- LinkedHashMap.Entry
- LinkedHashSet
- LinkedList
- List
- Long
- Map
- MarkupBuilder
- Math
- Random
- Set
- Short
- SimpleDateFormat
- String
- StringBuilder
- StringReader
- StringWriter
- SubList
- TimeCategory
- TimeZone
- TreeMap
- TreeMap.Entry
- TreeMap.KeySet
- TreeMap.Values
- TreeSet
- URLDecoder

- URLEncoder
 - UUID
 - XPath
 - XPathConstants
 - XPathExpressionImpl
 - XPathFactory
 - XPathFactoryImpl
 - XPathImpl
 - ZoneInfo
 - com.amazonaws.services.s3.model.S3Object
 - com.amazonaws.services.s3.model.S3ObjectInputStream
 - com.sun.org.apache.xerces.internal.dom.DocumentImpl
 - com.sun.org.apache.xerces.internal.dom.ElementImpl
 - groovy.json.JsonOutput
 - groovy.json.JsonSlurper
 - groovy.util.Node
 - groovy.util.NodeList
 - groovy.util.XmlParser
 - groovy.util.XmlSlurper
 - groovy.xml.XmlUtil
 - java.net.URI
 - java.util.RandomAccessSubList
 - org.apache.commons.codec.binary.Base64
 - org.apache.xerces.dom.DocumentImpl
 - org.apache.xerces.dom.ElementImpl
 - org.codehaus.groovy.runtime.EncodingGroovyMethods
 - org.json.JSONArray
 - org.json.JSONException
 - org.json.JSONObject
 - org.json.JSONObject.Null
-

Summary and next steps

Now that you understand how and why SmartThings restricts certain features of the Groovy programming language, it's time to dive deeper and write our first SmartApp! Head over to the [Writing Your First SmartApp](#) (page 175) and learn how easy it is to program the physical world.

Part VI

Writing Your First SmartApp

This tutorial will guide you through writing your first SmartApp. Once you've read through the *Groovy With SmartThings* (page 161), this should be your next stop.

Goals

At the end of this tutorial, you will know:

- How to create a SmartApp using the web-based IDE.
- The key components of a SmartApp.
- How to gather input from a user to configure the SmartApp.
- How to subscribe to changes in a device's state.
- How to control devices.
- How to schedule a SmartApp to execute in the future.
- How to use the Simulator to test your SmartApp.
- How to publish your SmartApp and install it on your mobile phone.
- How to achieve world domination, without even trying.

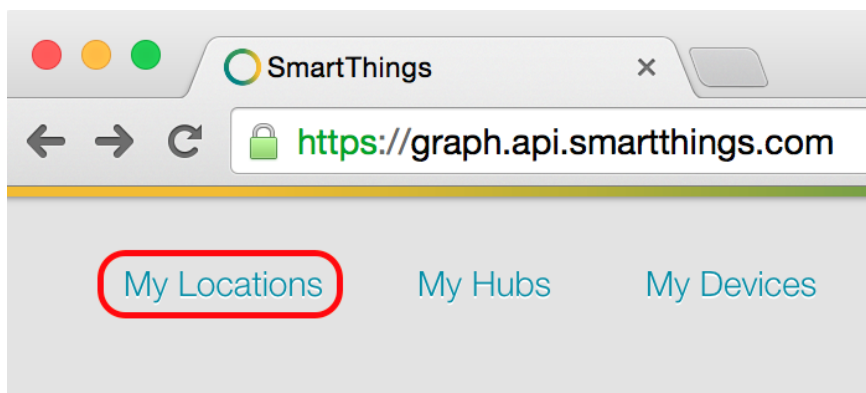
The SmartApp we will create will be relatively simple, but it will teach you a few core concepts of SmartThings, and get you familiar with the development process.

The purpose of the SmartApp we'll write is to turn a switch on when motion is detected, and turn it off when motion stops.

Prerequisites

Before completing this tutorial, you should have read the *Overview* (page 93), and registered for an account as discussed in the *Up and Running* (page 105) page. It is recommended that you become at least familiar with the basic Groovy concepts discussed in the *Groovy Basics* (page 115) and *Groovy With SmartThings* (page 161) tutorials.

Start by logging into IDE at <https://graph.api.smartthings.com>. Next, navigate to *My Locations* page to see the Locations you created.



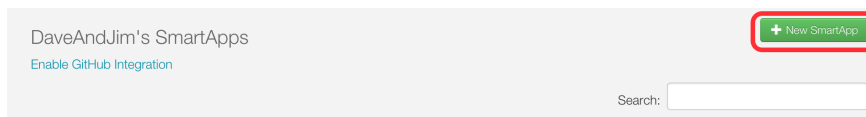
Normally you will see just one Location where you installed your Hub. Click on the Location name appearing in the far left column (i.e., the *Name* column). You may need to log in again with your SmartThings userid and password.

Warning: Note that even though the IDE is located at <https://graph.api.smartthings.com>, it may not always be the correct URL for your SmartApp deployment. By explicitly selecting the Location name you will ensure that your SmartApp will be published properly.

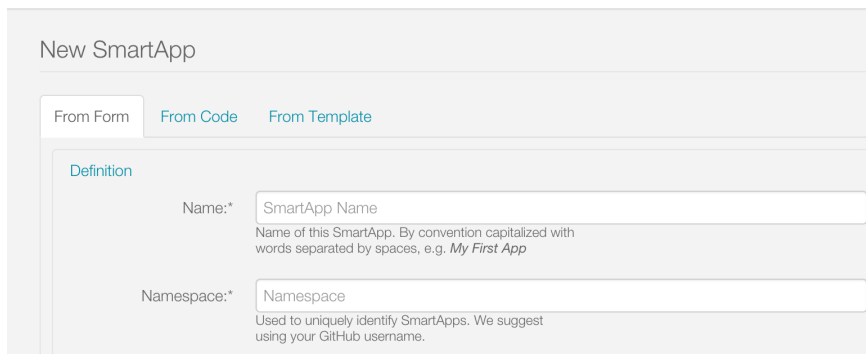
The SmartApp will utilize a motion sensor and a smart switch. Even if you don't have these devices or don't have a Hub, you can still complete the majority of this tutorial. We will call out any special steps required if you don't have the hardware.

Create a SmartApp

In the IDE, navigate to the *My SmartApps* page. This will bring you to a page that shows all of the SmartApps that you have created. This is also where you can create a new SmartApp. Click on the *New SmartApp* button.



Three options are presented for creating a new SmartApp: *From Form*, *From Code*, and *From Template*.



The *From Form* option will ask for some details about your SmartApp and create a SmartApp with some boiler plate code.

The *From Code* option will create a new SmartApp out of code that you paste into the input box.

Lastly, the *From Template* option will let you select an already existing SmartApp and use its code as a starting point. This is useful when you want to change or enhance a SmartApp that already exists, and it also a great way to look at examples.

For our SmartApp, let's stick to the *From Form* option.

Fill out the form as follows:

Name A name for your SmartApp. Call it something like “My First SmartApp”.

Namespace This field uniquely identifies your SmartApp in the event that someone else has written a SmartApp with the exact same name. This should be your GitHub username (or if you don't have a GitHub account, some other unique identifier).

Author This is you. Populate this field with your handle.

Description This describes the intent and functionality of your SmartApp. This description appears in the SmartApp Marketplace section of SmartThings mobile application, and hence a clear and concise description is recommended.

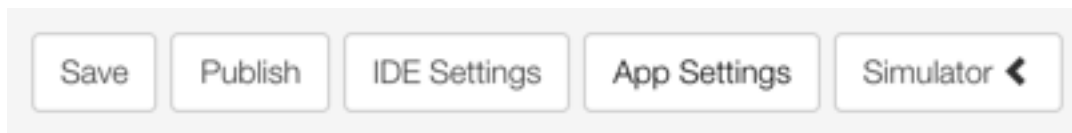
Category SmartApps are categorized based on functionality. This is used by SmartThings mobile application. SmartApps can be published either for Marketplace or for your own use. When publishing SmartApps for your own use (which is what we will be doing), all SmartApps will appear in *My Apps* category.

Leave the rest of the fields as they are, and click the *Create* button at the bottom. This will create the SmartApp and populate it with some skeleton code. In the next section we will dive into using the editor to begin writing your first SmartApp.

Editor

Once you've created your SmartApp, you'll be taken to the editor and Simulator. Before we look at the code, it's worth becoming familiar with some of the basic features.

Above the code window, there are five buttons:



Save This button saves your SmartApp in the SmartThings cloud.

Publish This allows you to publish your SmartApp for yourself, so you may install it in your SmartThings mobile app, as well as to submit it to the SmartThings team for publication into the SmartThings catalog.

IDE Settings Here you can make changes to personalize the editor to your liking. You can choose from a variety of themes to control the look and feel, specify your preferred keymapping, and set the font size.

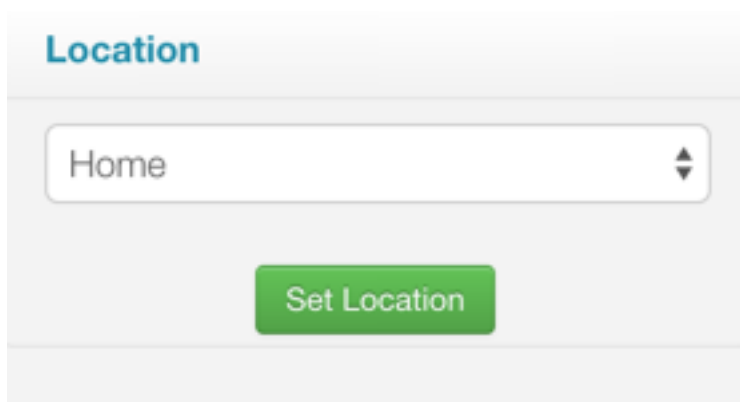
App Settings This takes you back to the form that you created this SmartApp from, where you can view the values entered when you created the SmartApp, as well as edit certain properties about the SmartApp.

Simulator This button toggles the display of the online Simulator. We'll discuss the Simulator in further detail next.

Tip: On the upper-right side of the IDE, in the *Simulator* menu, you'll see a drop-down titled *Browse SmartApp Templates*. If you click this, you'll see a variety of SmartApps that you can browse to learn from, or use as the starting point of a new SmartApp.

Simulator

On the right side of the IDE is the Simulator. This is where you can install your SmartApp to test it, either using physical devices, or simulated devices. We will walk you through installing the SmartApp using this later in the tutorial.



If you don't have a Location yet, the Simulator will show a message instructing you to create one. Follow the steps there to create a Location.

SmartApp basics

The first thing to know is that there are a few different types of SmartApps.

Some SmartApps, called *Service Manager SmartApps*, manage the connection of a Cloud-connected or LAN-connected device.

Solution Module SmartApps provide a dashboard-like user interface in the SmartThings mobile application ¹.

The most common type of a SmartApp is one that monitors the user's devices for certain changes (or simply execute on a defined schedule), and then take certain action ("Turn a light on when motion is detected"). These SmartApps are called *Event-Handler SmartApps*.

This tutorial will walk you through building a simple Event-Handler SmartApp, but the core principles you will learn are applicable to all types of SmartApps.

Regardless of what type of SmartApp you are writing, there are a few core principles that apply to all SmartApps:

- SmartApps are not continuously running. They are executed in response to various Events or schedules.
- SmartApps are installed into a user's Location, and a user may install multiple instances of a SmartApp into the same Location.
- With the exception of Solution Module SmartApps, SmartApps do not have any user interface, except for the preferences page that allows the user to configure the SmartApp (more on this in a bit).
- The code that defines a SmartApp does not run on the user's mobile phone. SmartApps may execute in the SmartThings cloud, or on the Hub. The mobile application uses some information from the SmartApp to drive the experience in the app.

In your editor, you can see that there is some code already written for you. This defines the basic structure and skeleton for your SmartApp. We will discuss each key component as we build our SmartApp.

¹ Solution Module SmartApps are not currently available for developers, but support for this is planned in the near future.

Definition

Every SmartApp must have a `definition` method call. This provides metadata about the SmartApp itself. The `definition` method simply expects a map of parameters. If you look at the code in the editor, you'll see that these values are already set from the values you entered when creating your SmartApp:

```
definition(  
  name: "My First SmartApp",  
  namespace: "mygithubusername",  
  author: "Peter Gregory",  
  description: "This is my first SmartApp. Woot!",  
  category: "My Apps",  
  imageUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",  
  iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",  
  iconX3Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png")
```

We don't need to change anything here, so let's move on to defining our preferences. If you do need to change some of your SmartApp's metadata, you can change these values later.

Preferences

The `preferences` method is where we define what information our SmartApp needs from the user. When a user installs a SmartApp on their mobile device, they will be taken to a screen (or screens) where they can configure the SmartApp. The content of these screens are derived from our `preferences` definition.

Preferences can be displayed as a simple, single screen, or multiple screens. This tutorial will use a simple preferences definition, with only one screen.

In the editor, there is a `preferences` definition stubbed in for us:

```
preferences {
  section("Title") {
    // TODO: put inputs here
  }
}
```

Recall that the purpose of our SmartApp is to turn a switch on when motion is detected. Our SmartApp needs to know which switch and motion sensor to work with. Update `preferences` with this code:

```
preferences {
  section("Turn on when motion detected:") {
    input "themotion", "capability.motionSensor", required: true, title: "Where?"
  }
  section("Turn on this light") {
    input "theswitch", "capability.switch", required: true
  }
}
```

Notice that we defined two `section` calls. Sections allow us to group related inputs, and can have a text description (“Select a switch to turn on”).

We use the `input` method to specify what types of devices we want the user to choose from. Let’s break down in detail the `input` for the switch:

```
input "theswitch", "capability.switch", required: true
```

The first argument to `input` is what we - inside our SmartApp - want to refer to the device as. In this case, we use `"theswitch"`. This becomes the identifier for the device in our SmartApp, so that we can refer to the switch as `theswitch` (without the quotes). We’ll see this in action shortly.

The second argument is the type of device our SmartApp will work with. `"capability.switch"` states that our SmartApp is requesting the user to pick from *any* device that supports the *Switch capability*. The concept of capabilities is core to SmartThings, and requires a bit more explanation.

First, consider that the catalog of connected devices is growing at a rapid pace. New devices arrive on the market almost daily. Many of these devices do similar things, and some do multiple things.

83.1 Capabilities

SmartThings abstracts devices into their *capabilities* - that is, what the device is capable of. This allows us to build SmartApps that can work with *any* device that supports a given capability. In this way, we can build robust SmartApps that will work with any device integrated with SmartThings that supports a given capability.

Capabilities are broken down into *commands* and *attributes*. *Commands* can be issued to a device, and *attributes* are what the device reports on. Every capability defines its commands and attributes, and devices that support a given capability must support those commands and attributes.

Note: A device may (and typically does) support multiple capabilities. For example, a Phillips Hue Bulb supports the Switch capability, because it can turn on and off. It also supports the Color Control capability, since the bulb can change colors. In our example, a Hue bulb could be selected by the user since it supports the Switch capability.

But, our SmartApp is only requesting that a user select a device that supports the Switch capability, so even if the user selects a device that can do more (such as a Hue bulb), we cannot assume that in our SmartApp. All we can know is that the device supports the Switch capability.

With capabilities, we can be assured that even if a new device supporting the Switch capability is added after we've written and published our SmartApp, there's no need to update any code!

Capabilities are created and maintained by SmartThings. You can view the reference documentation for capabilities in the [Capabilities Reference](#) (page 655).

The last thing to note in our `input` method call is the `required: true` argument. This specifies that the user must select a device in order to install the SmartApp.

Important: By requiring users to select which devices the SmartApp will work with, SmartThings is providing a basic security feature - SmartThings can only control those devices which a user explicitly chooses. SmartApps cannot control devices which the user did not select, and this is by design.

To summarize, when the user selects and installs the SmartApp from within SmartThings mobile app, they will be prompted to select a device that supports the switch capability. The SmartThings mobile app will provide them with a list of devices for this user's Location that support the switch capability. The device chosen will then be identified within the SmartApp as `theswitch`.

We covered a lot of information for such a small amount of code because it's important that you understand the importance of `preferences` and `capabilities`.

For additional information about preferences, see the [Preferences and Settings](#) (page 285) chapter of the SmartApp guide.

Now that you've updated the `preferences` method, make sure to save your SmartApp by clicking the *Save* button.

Events and callback methods

Our SmartApp needs to turn a switch on when motion is detected. To turn the switch on, we first need to know when motion is detected.

SmartApps can subscribe to various Events so that when that Event happens, the SmartApp will be notified. For our SmartApp we do this by using the *subscribe()* (page 942) method.

In your editor, below the preferences, you'll see some methods already defined:

```
def installed() {
  log.debug "Installed with settings: ${settings}"
  initialize()
}

def updated() {
  log.debug "Updated with settings: ${settings}"
  unsubscribe()
  initialize()
}

def initialize() {
  // TODO: subscribe to attributes, devices, locations, etc.
}

// TODO: implement event handlers
```

Every SmartApp must define methods named *installed()* (page 897) and *updated()* (page 898). When a user installs a SmartApp by clicking on the *Install* button in the SmartThings mobile application (after filling out any required preferences inputs), the *installed()* method we define in our SmartApp will be called. This is where SmartApps can subscribe to any device changes we are interested in, as well as set up any scheduled tasks we want our SmartApp to perform.

Similarly, the *updated()* method is called when a user updates their installation of the SmartApp by changing any of the preferences inputs. For example, a user may want to change which switch is turned on after they have installed the SmartApp. So, they open the SmartApp settings, select a different switch, and then update the SmartApp. At this point, the *updated()* method is called.

In our *updated()* method, notice that the first thing we do (aside from some logging, which is discussed shortly), is to call a method called *unsubscribe()* (page 947). This method is provided by the SmartThings platform, and simply removes any existing subscriptions this SmartApp has created. This is important, since the user has just changed their preferences for this SmartApp. If we didn't do this, we might still be subscribed to Events for devices that the user has removed from the SmartApp.

Also, note that both *installed()* and *updated()* call a method named *initialize()*. Since both *installed()* and *updated()* typically both create subscriptions or schedules, we can reduce code duplication

by using a helper method.

We also use the built-in logger (`log`) to log information. SmartThings does not currently have a debugger within the IDE, so use the `log()` method to log information that might be useful for debugging. The logs are available by clicking *Live Logging* at the top of the IDE.

Finally, note that we reference a variable named `settings` in our log statement. Remember the preference inputs we defined? Every preference input gets stored in a read-only map called `settings`. We can get the values of the various inputs by indexing into the `settings` map with the name of the input (e.g., `settings.theswitch`).

Now that you understand the purpose and importance of the `installed()` and `updated()` methods, we need to subscribe to any Events that we are interested in. In our case, we need to know when the motion sensor reports that it detected motion.

In the editor, update the `initialize()` method with this:

```
def initialize() {
    subscribe(themotion, "motion.active", motionDetectedHandler)
}
```

The `subscribe()` method accepts three parameters: The thing we want to subscribe to (`themotion`), the specific attribute and its state we care about (`"motion.active"`), and the name of the method (`motionDetectedHandler`) that should be called when this Event happens.

How do you know what attribute and what state we can subscribe to? We refer to the *Capabilities Reference* (page 655) to find out the available attributes the capability supports. In the case of the Motion Sensor capability, we see that it supports the `"motion"` attribute. In this case, it has two discreet possible values - “active” and “inactive”.

Since the `"motion"` attribute value is either active or inactive, we can subscribe to either of those specific changes by using the format `"<attribute>.<value>"`. This will cause the specified event handler method to be called any time the `"motion"` attribute value changes to `"active"` (motion is detected).

Now that we’ve created our subscription, we need to define the event handler method.

Event Handler methods

Add the following method to your SmartApp. We'll fill in the real meat of the method later.

```
def motionDetectedHandler(evt) {  
    log.debug "motionDetectedHandler called: $evt"  
}
```

Every event handler method must accept a single parameter, which is an *Event* (page 1017) object that contains information about the Event, such as the Event's value, time it occurred, and other information.

Since we subscribed to the "active" state of the motion sensor, we know that our event handler method will only be called when the motion sensor changes from inactive to active.

Now that we know motion has been detected, we need to turn the light on!

Controlling devices

Recall that capabilities support commands (things the device can do), as well as attributes (things the attribute knows). To turn the switch on requires only one line of code to be added to our event handler:

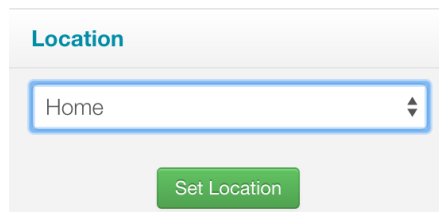
```
def motionDetectedHandler(evt) {  
  log.debug "motionDetectedHandler called: $evt"  
  theswitch.on()  
}
```

Simple, right? But how do we know that we can call the `on()` method on the switch? By looking at the Switch Capability Reference, we see that the Switch capability supports the `on()` and `off()` commands. These turn the switch on and off, respectively.

Also note that we referred to the switch selected by the user by the name we provided in the `input` inside preferences (`theswitch`).

Using the Simulator

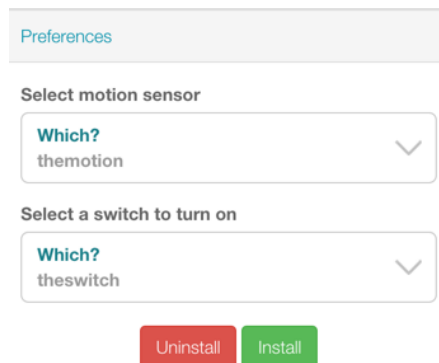
Save your SmartApp by clicking the *Save* button at the top of the IDE. Click *Simulator* and you will see a Location section on the right-hand side:



The screenshot shows a panel titled "Location" with a dropdown menu containing the text "Home" and a green "Set Location" button below it.

SmartApps are installed to a Location in your SmartThings account. By clicking the *Set Location* button, you are telling the Simulator that you want to install this SmartApp into the chosen Location.

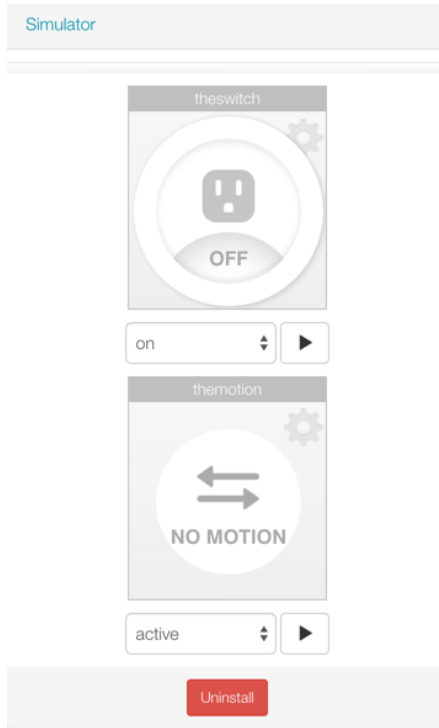
After you have selected the Location, you will see the *Preferences* section appear:



The screenshot shows a panel titled "Preferences" with two dropdown menus. The first is labeled "Select motion sensor" and has "Which?" and "themotion" selected. The second is labeled "Select a switch to turn on" and has "Which?" and "theswitch" selected. Below the dropdowns are two buttons: "Uninstall" (red) and "Install" (green).

This is where you can choose devices that the SmartApp will use. Here we see that it asks for a motion sensor to monitor, and a switch. These two inputs directly correspond to what we have in the `preferences` section in our SmartApp. SmartThings will provide a “Virtual Device” when it can. When you do not have a physical device to choose from this is a very useful option. By default the virtual devices will be selected. Click the *Install* button, and the SmartApp will be installed into the Location you selected above.

Now we see the Simulator section appear:



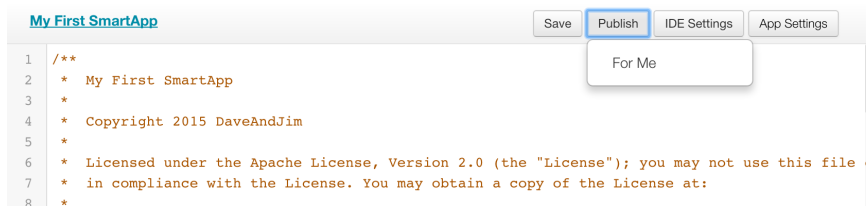
We have two devices. A motion sensor, and a switch. We can manipulate the motion sensor by choosing *active* or *inactive* and clicking the play button. The same with the switch, it can be *on* or *off*. We wrote our SmartApp to turn the switch on when motion is detected, so let's give that a try. Choose *active* if it's not already selected and then hit the play button. You should see the switch should go on:



Warning: The behavior of the Simulator is known to have inconsistencies. If you are unable to see the correct device status, or unable to actuate the device, you may just be experiencing issues with the Simulator. In that case, just skip ahead to the next section to install the SmartApp via the SmartThings mobile app.

Publishing and installing

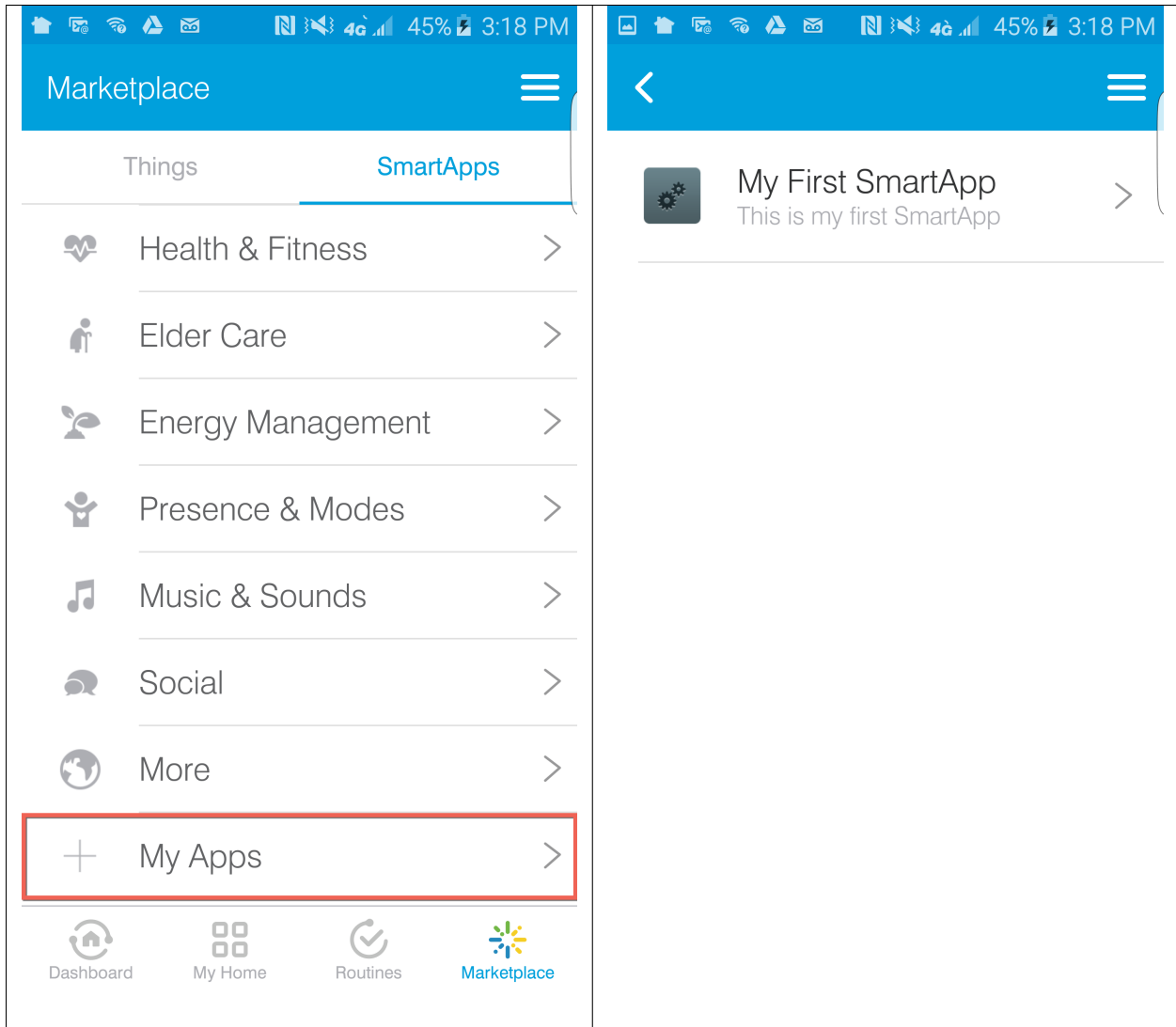
We can now see our first SmartApp in action in the Simulator. The next question is how can we use this SmartApp on our mobile devices in the SmartThings app? To accomplish this, we need to publish the SmartApp.



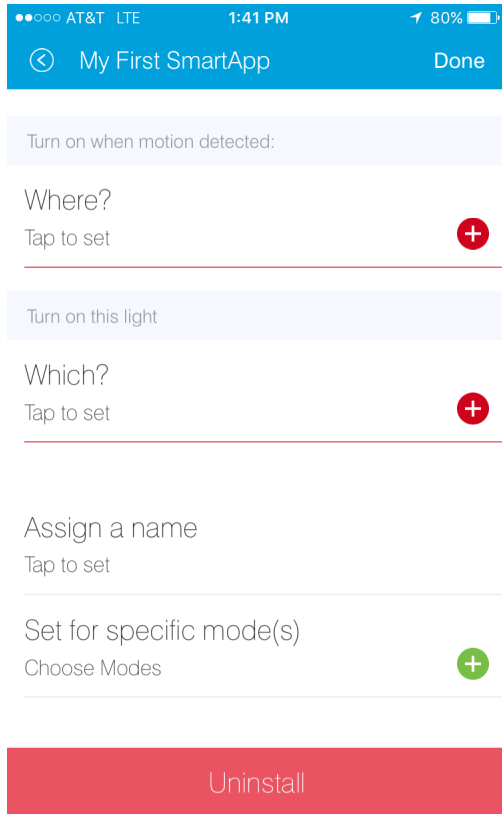
When you press the *Publish* button, a *For Me* option will appear. Select it. This means that the SmartApp will only be published for your account and not be visible for everyone in the SmartThings community.

Note: If you have a SmartApp that you do want to publish publicly, you can do that via the “My Publication Requests” link at the top of the page. For more information on this, see [For public distribution](#) (page 627).

Now you should be able to see your SmartApp in the mobile app if you browse to the *My Apps* category of the Marketplace:



After selecting your SmartApp, you will be brought to the preferences screen where you can select the devices to work with this SmartApp:



You can see the sections and inputs we defined in the `preferences` here. Notice how the inputs are marked in red, to indicate that the user must set values for these inputs in order to install the SmartApp.

Tap the fields to select a motion sensor and switch. If you have devices that support the requested capability, you'll see an option to select them.

You'll also see that some other inputs were added for us. For single page preferences, every SmartApp receives an input to allow the user to assign a name of their choosing for this installation. The name that they choose will then be displayed as the name of the SmartApp. Also by default, the user can select to only execute this SmartApp when the Location is in certain *Modes* (page 335). It also includes the ability for the user to uninstall this SmartApp.

Note: A SmartApp may be installed into a Location multiple times. For example, a person may have multiple rooms for which they want a light to come on when motion is detected.

Even though the code is the same, each installation is unique, and must also be removed by the user individually.

Turn off when motion inactive

We now have a simple SmartApp that turns a switch on when motion is detected. Let's extend this further, and turn the switch off when the motion stops.

In our SmartApp, we need to subscribe to not only the motion sensor being active, but also inactive.

Recall that our subscription looks like this:

```
subscribe(themotion, "motion.active", motionDetectedHandler)
```

We will also subscribe the "motion.inactive" Event in a similar way. Add this subscription to the initialize() method:

```
subscribe(themotion, "motion.inactive", motionStoppedHandler)
```

Note: We could also subscribe to *any* change in the motion sensor, by simply specifying the attribute we want to monitor (e.g., "motion" instead of "motion.active"). This would then call the specified handler method when there is any reported change to the "motion" attribute. For attributes that don't have a discrete set of possible values (for example, temperature readings), this is how we subscribe to changes for that attribute.

We can then get the value of the Event in the event handler by looking at the value of the passed-in Event. If we were to do this in our SmartApp, it would look like this:

```
def initialize() {
    subscribe(themotion, "motion", motionHandler)
}

def motionHandler(evt) {
    if (evt.value == "active") {
        // motion detected
    } else if (evt.value == "inactive") {
        // motion stopped
    }
}
```

Our SmartApp will use separate subscriptions and event handlers, but you are free to modify it to use a single subscription and handle the different values in your event handler method.

We need to define the motionStoppedHandler event handler method - add this method to your SmartApp:

```
def motionStoppedHandler(evt) {
    log.debug "motionStoppedHandler called: $evt"
    theswitch.off()
}
```

Save your SmartApp in the IDE, publish it again for yourself, and then install it again in the Simulator. Now when you change the motion to “inactive”, the switch will turn off.

Going further—adding flexibility

Our SmartApp now turns a switch on when motion is detected, then turns it off when motion stops. But consider this scenario:

- A person enters a room, the motion sensors reports that motion is active, and our SmartApp turns the light on.
- The person then sits down, or stands still enough for the motion sensor to report motion is inactive, and our SmartApp turns the light off.
- The person than moves again, causing the motion sensor to again report active motion, and our SmartApp turns the light on again.

As you can imagine, this could be quite annoying. It would be better if we could allow the user to specify a number of minutes *after motion stops* to turn the light off. Then, once motion stops, if no motion is detected within the specified number of minutes, the SmartApp will turn the light off. If motion is detected within this time window, the switch will not turn off.

We can add this flexibility into our SmartApp easily. The first thing we need to do is update our `preferences` to let the user specify the number of minutes to elapse without motion being detected, before the light is turned off.

Replace the `preferences` in our SmartApp with the following:

```
preferences {
  section("Turn on when motion detected:") {
    input "themotion", "capability.motionSensor", required: true, title: "Where?"
  }
  section("Turn off when there's been no movement for") {
    input "minutes", "number", required: true, title: "Minutes?"
  }
  section("Turn on/off this light") {
    input "theswitch", "capability.switch", required: true
  }
}
```

Preferences inputs can be more than just devices - we can ask users to enter in numeric values, text values, booleans, enumerated lists, and more. You can learn about the various options for preferences inputs [here](#) (page 285).

Now that the user can specify the number of minutes to wait without motion before turning the light off, we need to implement the logic to do so.

Our `motionStoppedHandler()` method will be called whenever the motion sensor reports that motion has stopped. Before turning the light off, we need to check that there is no motion detected for the specified number of minutes in the future. But since SmartApps are not continuously running, how can we handle checking for future states? The answer is by using methods that allow us to schedule a SmartApp for future execution.

The first thing we need to do is update our `motionStoppedHandler()` to execute a method after the number of minutes specified by the user. This method will then check to see if there has been motion reported within the time interval, and turn the light off if there has been no motion.

Let's write some skeleton code to do this, and we'll fill in the details later. First, update the `motionStoppedHandler()` method and add a new method as shown below:

```
def motionStoppedHandler(evt) {
    log.debug "motionStoppedHandler called: $evt"
    runIn(60 * minutes, checkMotion)
}

def checkMotion() {
    log.debug "In checkMotion scheduled method"
}
```

We use the `runIn()` (page 928) method to schedule our `checkMotion()` method to be called after the number of minutes specified by the user. We pass `runIn()` the number of seconds (from the time of the call) to schedule the call, and the name of the method we want executed.

When motion stops, our `checkMotion()` method will be called after the number of minutes specified by the user. Now, inside our `checkMotion()` method, we need to see if there has been any motion detected in the time window specified. We can use some date/time utility methods, along with information about the device state, to determine if we should turn the switch off.

Here's the logic we need to implement:

- If the motion sensor is currently reporting active motion, do nothing.
- If the motion sensor is reporting inactive motion, check to see what time the motion sensor reported inactive motion.
- If the motion sensor reported that motion has been inactive for longer than the time specified by the user, turn the switch off.

And here's the full method definition for `checkMotion()`. Update your SmartApp with the code below:

```
def checkMotion() {
    log.debug "In checkMotion scheduled method"

    // get the current state object for the motion sensor
    def motionState = themotion.currentState("motion")

    if (motionState.value == "inactive") {
        // get the time elapsed between now and when the motion reported inactive
        def elapsed = now() - motionState.date.time

        // elapsed time is in milliseconds, so the threshold must be converted to milliseconds too
        def threshold = 1000 * 60 * minutes

        if (elapsed >= threshold) {
            log.debug "Motion has stayed inactive long enough since last check ($elapsed ms): turning
            theswitch.off()
        } else {
            log.debug "Motion has not stayed inactive long enough since last check ($elapsed ms): do
        }
    } else {
        // Motion active; just log it and do nothing
        log.debug "Motion is active, do nothing and wait for inactive"
    }
}
```

The first thing to note is that we get a *State* (page 1040) object for the motion sensor, by using the `currentState()` method with "motion" as the attribute we're interested in. This object encapsulates information about an attribute at a particular moment in time. In our case, we want the current state.

From this object, we can determine when this state record was created. This will be the time that the motion sensor reported it is inactive. Using the *now()* (page 926) method, we can get the current time (in milliseconds), and then see if the motion stopped within the threshold specified by the user. If the time elapsed since the motion stopped exceeds the threshold, we turn the switch off.

Go ahead and save and publish your SmartApp again, and try it out!

Complete code listing

Here is the entire code for our SmartApp:

```
definition(  
  name: "My First SmartApp",  
  namespace: "mygithubusername",  
  author: "Peter Gregory",  
  description: "This is my first SmartApp. Woot!",  
  category: "My Apps",  
  imageUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",  
  iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",  
  iconX3Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png")  
  
preferences {  
  section("Turn on when motion detected:") {  
    input "themotion", "capability.motionSensor", required: true, title: "Where?"  
  }  
  section("Turn off when there's been no movement for") {  
    input "minutes", "number", required: true, title: "Minutes?"  
  }  
  section("Turn on this light") {  
    input "theswitch", "capability.switch", required: true  
  }  
}  
  
def installed() {  
  initialize()  
}  
  
def updated() {  
  unsubscribe()  
  initialize()  
}  
  
def initialize() {  
  subscribe(chemotion, "motion.active", motionDetectedHandler)  
  subscribe(chemotion, "motion.inactive", motionStoppedHandler)  
}  
  
def motionDetectedHandler(evt) {  
  log.debug "motionDetectedHandler called: $evt"  
  theswitch.on()  
}
```

```
def motionStoppedHandler(evt) {
    log.debug "motionStoppedHandler called: $evt"
    runIn(60 * minutes, checkMotion)
}

def checkMotion() {
    log.debug "In checkMotion scheduled method"

    def motionState = themotion.currentState("motion")

    if (motionState.value == "inactive") {
        // get the time elapsed between now and when the motion reported inactive
        def elapsed = now() - motionState.date.time

        // elapsed time is in milliseconds, so the threshold must be converted to milliseconds too
        def threshold = 1000 * 60 * minutes

        if (elapsed >= threshold) {
            log.debug "Motion has stayed inactive long enough since last check ($elapsed ms): turning
                theswitch.off()
        } else {
            log.debug "Motion has not stayed inactive long enough since last check ($elapsed ms): do
        }
    } else {
        // Motion active; just log it and do nothing
        log.debug "Motion is active, do nothing and wait for inactive"
    }
}
```

How the switch turns on (or off)

Now that we understand how to control devices in a SmartApp, you may be wondering how exactly the method `switch.on()` turns on the switch. The answer is Device Handlers.

Device Handlers are software much the same way SmartApps are. They define what actually happens when you call `switch.on()`. Let's look at an example to further understand this.

When you connect a new device to your SmartThings Hub, a Device Handler is picked for it based on the signature the device delivered to the Hub as part of its pairing communication. The Device Handler will have methods defined in it that support that device. So in our case, the Device Handler for the specific switch being used will have both `on()` and `off()` methods defined. The actual implementation of these methods vary depending upon the underlying device protocols, but are typically low-level protocol-specific commands to send to the device (like Z-Wave or ZigBee).

So, when `switch.on()` is executed from your SmartApp, the SmartThings platform will look up the Device Handler associated with the device and call its `on()` method, which will in turn send the protocol and device-specific command through the Hub to the device. Device Handlers are discussed in the [Device Handlers](#) (page 449) guide.

Summary

In this tutorial, you learned how to write a SmartApp. To do this, we:

- Created a new SmartApp using the web-based IDE.
 - Defined the `preferences` that specifies what input we need from the user.
 - Subscribed to device Events and controlled a device. We used the *Capabilities Reference* (page 655) to determine what attributes and commands a capability supports.
 - Used the web-based Simulator to test our SmartApp with virtual devices.
 - Published the SmartApp for yourself and installed it on your mobile phone.
 - Extended our SmartApp by allowing a user to enter the number of minutes to wait before turning the switch off, and implemented this using the `runIn ()` method.
-

Next steps

Now that you've written your first SmartApp and have a basic understanding of the SmartThings developer tools, language, and workflow, here are some further topics for you to pursue.

94.1 More about SmartApps

There is much more you can do with SmartApps than what this tutorial covered. SmartApps can *send notifications* (page 385), *execute routines* (page 339), *define advanced schedules* (page 345) for which they execute, *call external web services* (page 365), and more. You can learn more about developing SmartApps in the *SmartApps* (page 279) guide.

You can also make your SmartApp into a web service, capable of exposing its own REST endpoints. You can read about them in the *Web Services SmartApps* (page 411) guide.

94.2 Fork it!

SmartThings SmartApps and Device Handlers are now hosted in GitHub. Further, the IDE can integrate with GitHub, to provide a seamless developer experience. Learn more about it in the *GitHub Integration* (page 265) chapter of the *Tools and IDE* (page 251) guide. Happy forking!

94.3 Device Handler development

If you are interested in learning more about Device Handlers, and how to write one, head over to the *Device Handlers* (page 449) guide.

Part VII

Getting Help

In addition to this documentation, there are other ways to learn and get help developing for SmartThings, discussed below.

Developer documentation

Use this documentation to learn about SmartThings development, as well as serve as a reference. The documentation is searchable, and can be viewed and downloaded in a variety of formats including PDF and EPUB (click on “Read the Docs” on the bottom left of the page to see the download options).

This documentation is open source and available in GitHub [here](#).

Community

One of the best things about SmartThings is the amazing [community](#) of users, makers, and developers. Make sure to register for an account and introduce yourself. It's a great place to learn, help others, and make friends.

If you can't find an answer for your question in the documentation, the community is a great resource as well. You can search for your question in case it has already been addressed, or post a new question.

Many of the SmartThings staff frequents these forums as well. We'll chime in and try to be helpful.

SmartThings developer support

While our [community](#) is amazing and there are tons of awesome people there to support you, you may sometimes have more in-depth questions that our community can't answer. When that happens, we're here to help.

[This form](#) is a direct line to our developer advocates in the rare occasion our community can't help with your question. Once you submit a ticket you should expect to get a response in 48 hours.

Note: This form is for developers writing SmartApps and Device Handlers on the platform. If you need support for a SmartApp or Device Handler that you found on our community, please reach out to the developer of that SmartApp or Device Handler for support. If they need further support we will work directly with them to support their development.

In order to receive the best support, you should provide a **simplified** example that clearly illustrates the issue. This should be in the form of a simple SmartApp or Device Handler that can be easily installed and clearly shows the issue. This allows us to quickly verify the issue, and use it as a test for any fix provided.

We recommend creating a [gist](#) with a complete and easily installable SmartApp or Device Handler, and referencing it in your support ticket.

Important: We recognize that sometimes, providing a simple SmartApp or Device Handler that illustrates the issue is not possible. More often than not, however, it is. A simplified example enables you (and us) to verify that the issue is in fact with the API in question, and not some other factor.

If you do not provide a simplified example that can be easily installed, our ability to quickly verify, diagnose, and address the issue may be limited.

Part VIII

Architecture

As a starting point in understanding SmartThings approach, it is important to recognize that it is centered on the separation of *intelligence* from devices. SmartThings architecture is developed with a view that most of the value will be created in the *space between the devices*. Moreover, the devices themselves can be limited to their primitive capabilities (open/close, on/off, heat/cool, brew/don't brew), while the intelligence layer exists separately as an application layer.

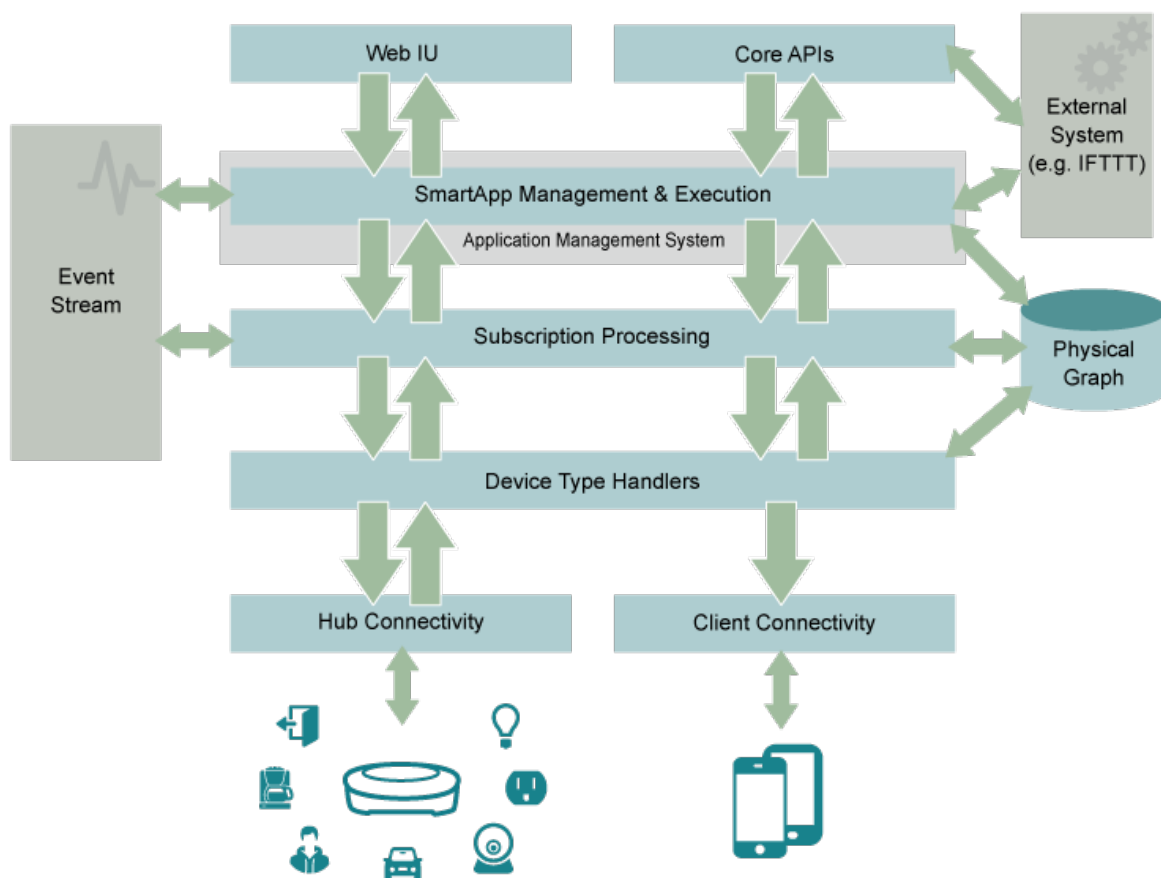
By doing this we allow the intelligence (or application) layer to apply flexibly across a wide range of devices, and make it easier to create applications that interact with and across the physical world. In many cases, we also benefit from lower-cost end devices, less maintenance complexity and longer battery life.

The SmartThings platform provides methods such that using these methods in Device Handlers we can abstract away the underlying complexity of devices and protocols, while at the same time coding in only the desired experience into SmartApps.

Each device in SmartThings has “capabilities”, which define and standardize available attributes and commands for a device. This allows you to develop an application for a type of device, regardless of the connection protocol or the manufacturer.

All of the code that developers can write on our platform is written in Groovy, which is a dynamic, object-oriented language built for the Java platform. You can learn more about Groovy on the [Groovy Basics](#) (page 115) page.

Big picture



98.1 Devices

Devices are the building blocks of the SmartThings infrastructure. They are the connection between the SmartThings system and the physical world. There's a huge variety in the devices you can use; some are created by SmartThings, but most are not.

The real power of SmartThings is that the platform works with most home automation devices already on the market. We believe in a fully integrated approach, where you aren't tied into a particular technology or protocol. SmartThings

offers compatibility with standards such as ZigBee, Z-Wave, LAN, and Cloud-to-cloud integrations. This allows SmartThings platform to work with hundreds of off the shelf third-party devices.

98.2 Hub

The SmartThings Hub connects directly to your broadband router. The Hub provides communication between all connected devices, the SmartThings cloud and the SmartThings mobile application. With a SmartThings Hub you:

- Simply plug it into your Ethernet router and provide power.
- Connect any SmartThings or SmartThings-ready device to your SmartThings account.
- Build your own SmartThings kit by combining with other SmartThings devices.
- Work with a variety of standard ZigBee and Z-Wave devices, such as GE Z-Wave in-wall switches and outlets.

The new Samsung SmartThings Hub also supports the ability to execute certain automations locally on the Hub itself, and ships with four AA batteries. This allows for certain automations to continue, even without AC power. It also ships with USB ports and is Bluetooth Low Energy capable. While not active at launch, this allows for greater expansion in the future without requiring new hardware.

98.3 Connectivity management

Connectivity Management is the layer that connects your SmartThings Hub, the client devices (mobile phones) to SmartThings servers and to the cloud as a whole. The Connectivity Management layer is comprised of:

- Hub Connectivity that connects your Hub to the cloud.
- Client Connectivity that connects your client devices to the cloud.

These are the highways by which your messages are sent to the internet.

98.4 Device Handler execution

The SmartThings system determines what type of device you are using based on Device Handlers. Once the Device Handler is selected, the incoming messages are parsed by that particular Device Handler. The input to the Device Handler is a set of device-specific messages, and the output of the Device Handler is normalized SmartThings Events. Note that one message can lead to many SmartThings Events.

98.5 Subscription management

When Events are created in the SmartThings platform, they don't inherently do anything besides publish that they've happened. Instead of Events triggering change, SmartApps are configured with subscriptions that listen for defined Events. The purpose of the subscription management layer is to match up Events that are triggered by the Device Handlers with the SmartApp that is using them.

98.6 SmartApp execution

The SmartApp is run when triggered either via subscriptions, or via external calls to SmartApp endpoints, or by scheduled methods. The SmartApp is transient in nature, as it runs and then stops running on completion of its task.

Any data that needs to persist throughout SmartApp instances must be stored in a special `state` variable that is discussed in the *Storing Data With State* (page 315) documentation.

98.7 Web UI and IDE

The Web UI sits on top of all of the other technology and allows you to monitor your devices, Hubs, Locations and many other aspects of your SmartThings system.

You have full control of the configuration, including editing, adding, removing, and even creating SmartApps. To create, you write code within the IDE for SmartApps and Device Handlers. SmartThings also has an integrated Simulator that allows you to simulate any devices, so it's not required to own the devices you develop for.

Important concepts

99.1 Asynchronous and eventually consistent programming

When dealing with the physical graph, i.e., a digital representation of the physical things connected around us, there will always be a delay between when you request something to happen and when it actually happens. There is latency in all networks, but it's especially pronounced when dealing with the physical graph.

To deal with this, the SmartThings platform utilizes asynchronous execution. This means that anytime you execute a command, it doesn't stop everything else from running. This helps everyone's code run the most efficiently.

Our basic methodology towards executing a command, such as turning a light switch on, is "fire and forget". This means that you execute a command, and assume it will turn on in due time, without any sort of follow up.

You cannot be guaranteed that your command has been executed, because another SmartApp could interact with your end device, and change its state. For example, you might turn a light switch on, but another app might sneak in and turn it off.

If you need to know if a command was executed, you can subscribe to an Event triggered by the command you executed and check its timestamp to ensure it fired after you told it to. You will, however, still have latency issues to take into consideration, so it's impossible to know the exact current status at any given time.

The SmartApps platform follows eventually consistent programming, meaning that responses to a request for a value in SmartApps will eventually be the same, but in the short term they might differ.

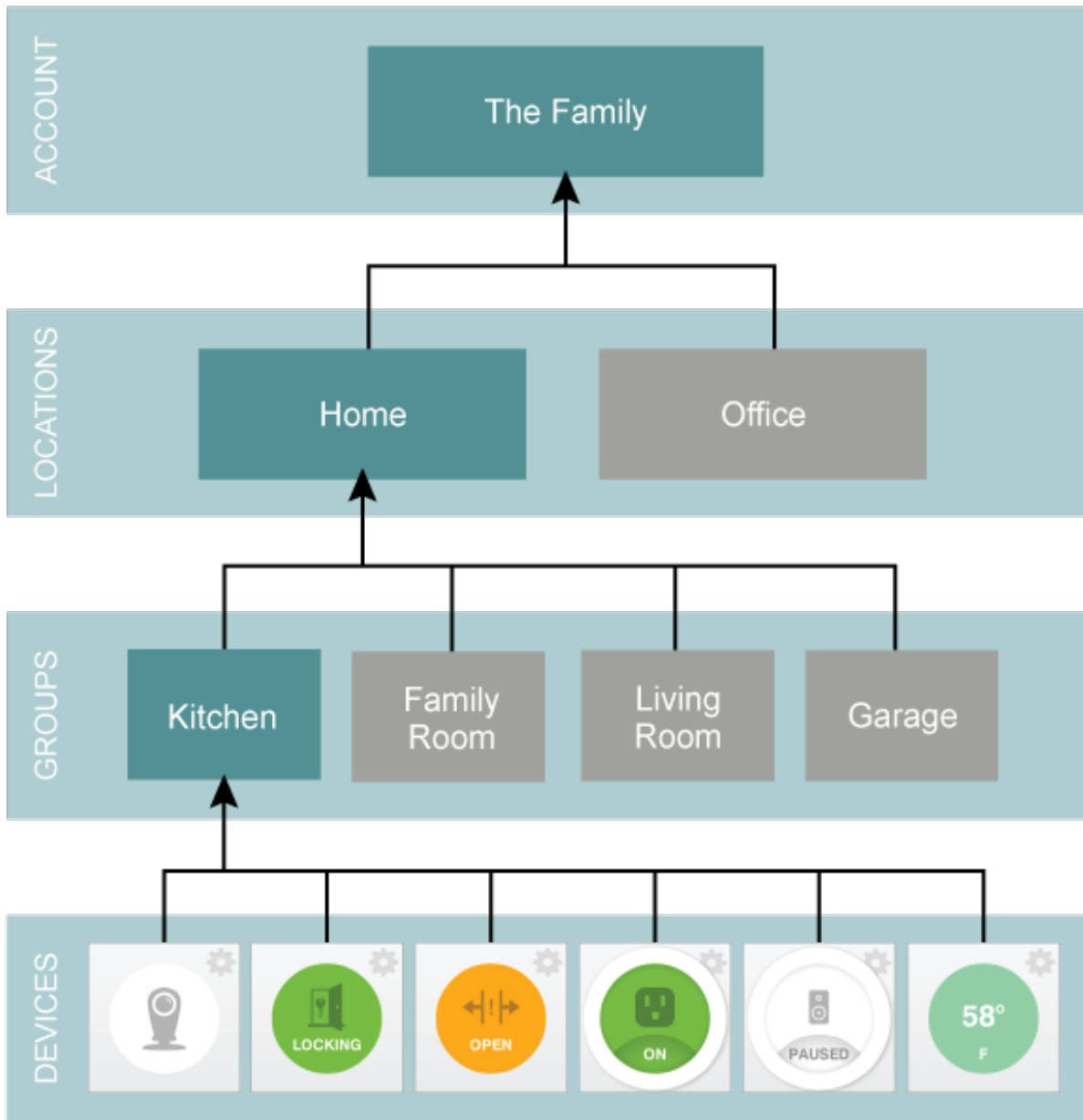
99.2 Containers

Within the SmartThings platform, there are three different "containers" that are important concepts to understand. These are: *accounts*, *Locations*, and *groups*. These containers represent both security boundaries and navigation containers that make it easy for users to browse their devices.

The diagram below shows the hierarchical relationship between these containers. Each type of container is described below in more detail.

99.3 Accounts

Accounts are the top-level container that represents the SmartThings 'customer'. Accounts contain only Locations and no other types of objects.



99.4 Locations and users

Locations are meant to represent a geolocation such as “Home” or “Office”. Locations can optionally be tagged with a geolocation (latitude and longitude). In addition, Locations don’t have to have a SmartThings Hub, but generally do. Finally, locations contain Groups or Devices.

99.5 Groups

Groups are meant to represent a room or other physical space within a Location. This allows for devices to be organized into groups making navigation and security easier. A group can contain multiple devices, but devices can only be in a single group. Further, nesting of groups is not currently supported.

Capability taxonomy

Capabilities represent the common taxonomy that allows SmartThings platform to link SmartApps with Device Handlers. An application interacts with devices based on their capabilities, so once we understand the capabilities that are needed by a SmartApp, and the capabilities that are provided by a device, we can understand which devices (based on the type of device and inherent capabilities) are eligible for use within a specific SmartApp.

The *Capabilities Reference* (page 655) is evolving and is heavily influenced by existing standards like ZigBee and Z-Wave.

Capabilities themselves may be decomposed into both ‘Actions’ or ‘Commands’ (these are synonymous), and Attributes. Actions represent ways in which you can control or actuate the device, whereas Attributes represent state information or properties of the device.

100.1 Attributes and events

Attributes represent the various properties or characteristics of a device. Generally speaking device attributes represent a current device state of some kind. For a temperature sensor, for example, ‘temperature’ might be an attribute. For a door lock, an attribute such as ‘status’ with values of ‘open’ or ‘closed’ might be a typical.

100.2 Commands

Commands are ways in which you can control the device. A capability is supported by a specific set of commands. For example, the ‘Switch’ capability has two required commands: ‘On’ and ‘Off’. When a device supports a specific capability, it must generally support all of the commands required of that capability.

100.3 Custom capabilities

We do not currently support creating custom capabilities. You can, however, create a device-type handler that exposes custom commands or attributes.

SmartThings cloud

The SmartThings platform assumes a “Cloud First” approach. This means that in order to use all supported devices and automations, and to ensure that the SmartThings mobile application reflects the correct state of your home, the SmartThings Hub will need to be online and be connected to the SmartThings cloud.

The second generation Hub, the Samsung SmartThings Hub, allows for some Hub-local capabilities. Certain automations can execute even when disconnected from the SmartThings cloud. This allows SmartThings to improve performance and insulate the user from intermittent internet outages.

This is accomplished by delivering certain automations to the Samsung SmartThings Hub itself, where it can execute locally. The engine that executes these automations are typically referred to as “AppEngine”. Events are still sent to the SmartThings cloud - this is necessary to ensure that the SmartThings mobile application reflects the current state of the home, as well as to send any notifications or perform other cloud-based services.

The specific automations that execute locally are expanding and currently managed by the SmartThings internal team. The ability for developers to execute their own SmartApps or Device Handlers locally is planned.

That said, there are a number of important scenarios where the cloud is simply required:

Scenario: There may not be a hub at all

Many devices are now already connected devices, via Wi-Fi/IP, and connect directly to the cloud without the need for a gateway device (hub).

The most likely use case for such devices involves adding intelligence to those devices through SmartApps. These devices may not be connected to a SmartThings Hub, and instead are directly connected to the vendor cloud or the SmartThings Cloud.

Put simply, if there is no Hub, then the SmartApps layer must run in the cloud!

Scenario: SmartApps may run across both cloud- and Hub-connected devices

As a corollary to the first point above, since there are use cases where devices are not Hub-connected, SmartApps might be installed to use one device that is Hub-connected, and another device that is Cloud-connected, all in the same app. In this case, the SmartApp needs to run in the cloud.

Scenario: There may be multiple Hubs

While the mesh network standards for ZigBee and Z-Wave generally eliminate the need for multiple SmartThings Hubs, we didn’t want to exclude this as a valid deployment configuration for large homes or even business applications of our technology. In the multi-Hub case, SmartApps that use multiple devices that are split across hubs will run in the cloud in order to simplify the complexity of application deployment.

Scenario: External service integration

SmartApps may call external web services. Calling them from SmartThings cloud reduces risk as it allows SmartThings to easily monitor for errors and ensure the security and privacy of the users.

In some cases, an external web service might even use IP white-listing such that they simply can't be called from the Hub running at a user's home or place of business.

Accordingly, SmartApps that use web services will run in the cloud also.

Important: Note that because of the abstraction layer, SmartApp developers never have to understand where or how devices connect to the SmartThings platform. All of that is hidden from the developer so that whether a device (such as a Garage Door opener) is Hub-Connected or Cloud-Connected, all they need to understand is:

```
myGarageDoor.open()
```

Hubs and Locations

To efficiently manage performance, the SmartThings platform scales its cloud server architecture horizontally with *sharding*. Sharding helps reduce the latency between the Hub and the cloud, and handles increasing capacity. As a developer you must note the impact of sharding on how you work with the SmartThings IDE.

When you first install SmartThings app on your mobile phone, create your user account and claim your Hub, the SmartThings platform automatically assigns your Hub to the Location and connects your Location/Hub to a particular shard. Before starting your development, you must note that:

- Your Location/Hub is connected to a *specific* SmartThings shard, based on the geographical location of the Hub, and,
- You must ensure that you are logged into the URL of this specific shard on IDE. Since the Location is always connected to the correct shard URL, you can do this by clicking on your Location from “My Locations” page after you log in.

Note: If for some reason you are not seeing your Hub in the IDE, then from *My Locations* page select the Location and it will prompt you to log into correct shard where you can see your Hub.

102.1 Consequences of sharding

In practice, some consequences of sharding are:

- A global layer, with a few specific services, spans across all shards while all other services are owned by the specific shard itself (which, as emphasized above, is Location-dependent). A few global layer services are: user account creation, authorization, OAuth authentication, mappings of Location-to-shard, users-to-Locations and Hub-to-Locations. All data that is down from the Location level are managed by the specific shard.
- A shard does not share information with another shard. For example, a common login across the shards does not exist yet. You will have to log in to each shard, although the userid and password will be the same (see the note above). At the same time, note that SmartThings mobile app users do not have to log in again because mobile client OAuth tokens are shared across the shards.
- SmartApps and Device Handlers are now published in a specific shard and not for your entire account. For example, if you have a Hub in North America and another Hub in Europe, you will need to publish your SmartApp twice, one in each Location, i.e., shard.
- Note that since a Hub is assigned to a Location, if you delete a Location, the Hub becomes unclaimed. Conversely, it is possible for a Location to exist without a claimed Hub at that Location.

Part IX

Tools and IDE

The [SmartThings IDE](#) (Integrated Development Environment) provides SmartThings developers with a set of tools to manage their SmartThings account, and build and publish custom SmartApps and Device Handlers.

Account Management

The SmartThings IDE allows you to view and edit information about your Locations, Hubs, Devices, custom SmartApps and Device Handlers, as well as view a live log for all your SmartThings devices and apps.

103.1 Locations

Name	Account	Groups			
Minneapolis Office	mplis@smarthings.com's Account	<ul style="list-style-type: none"> Lounge Hallway & Stairs Small Conference Room SmartBlock People Maker Lab Unassigned Main Suite Music Kitchen Bean Bag Room 	events	notifications	smartapps

My Locations will show all Locations registered to your account. Choosing a particular Location will allow you to see more in depth information on that Location, including the groups created under that Location. You can also see all Events, notifications, and SmartApps under a particular Location.

103.2 Hubs

My Hubs will show all Hubs registered to your account. Choosing a particular Hub will give a comprehensive look at all of the attributes of your Hub, with the opportunity to observe all Events that have taken place, by clicking on *List Events*. You can also view all of the devices that are registered to your Hub.

103.3 Devices

Display Name	Type	Location	Hub	Group	Zigbee Id	Device Network Id	Status	Last Activity
Adam's Android	Mobile Presence	Minneapolis Office		People			INACTIVE	3 weeks ago
Remote	Aeon Minimote	Minneapolis Office	Minneapolis Office Hub	Maker Lab			INACTIVE	2 weeks ago
Lounge Aeon Multisensor	Aeon Multisensor	Minneapolis Office	Minneapolis Office Hub	Lounge			INACTIVE	3 weeks ago
Andrew's Android	Mobile Presence	Minneapolis Office		People			INACTIVE	37 minutes ago
Beanbag Lamp	Dimmer Switch	Minneapolis Office	Minneapolis Office Hub	Bean Bag Room			INACTIVE	1 week ago
Main Suite	Dimmer Switch	Minneapolis Office	Minneapolis Office Hub	Main Suite			ACTIVE	2 hours ago
Kegerator Dropcam	Dropcam	Minneapolis Office		Lounge			INACTIVE	27 minutes ago
Kitchen Flood Detector	Everspring Flood Sensor	Minneapolis Office	Minneapolis Office Hub	Kitchen			INACTIVE	25 minutes ago
Hue Bridge	Hue Bridge	Minneapolis Office	Minneapolis Office Hub	Main Suite			INACTIVE	25 minutes ago
Hue Lamp 2	Hue Bulb	Minneapolis Office	Minneapolis Office Hub	Main Suite			ACTIVE	3 hours ago
Hue Lamp 1	testHue	Minneapolis Office	Minneapolis Office Hub	Main Suite			INACTIVE	18 minutes ago

My Devices will show all devices attached to any of your Hubs. Choosing a particular device will give a comprehensive look at all of the attributes of your device, with the opportunity to observe all Events that have taken place, by clicking on *List Events*.

103.4 SmartApps

My SmartApps will show all your custom (written or edited by you) SmartApps. You can view the SmartApp status, category, and Locations from this list, as well as edit SmartApp metadata. You can click the SmartApp name to be taken to the editor where you can view and modify the code.

103.5 Device Handlers

My Device Handlers will show all your custom (written or edited by you) Device Handlers. You can view the status, supported capabilities, and sessions from this list, as well as edit the metadata associated with this Device Handler. You can click on the name to be taken to the editor, where you can view and modify the code.

103.6 Publication requests

My Publication Requests will show all your publication requests for submissions to the SmartThings catalog, along with the publication request status.

103.7 Live logging

Live Logging will show a live logging view for your SmartThings account. Here you will find logs for all your installed SmartApps and Device Handlers. You can also filter the logs by a specific SmartApp.

Editor and Simulator

The SmartThings editor and simulator allows you to create, edit, and test SmartApps and Device Handlers.

The screenshot displays the SmartThings IDE interface. At the top, there are navigation tabs: My Locations, My Hubs, My Devices, My SmartApps, My Device Types, Logs, and Documentation. A 'Browse SmartApps' dropdown is on the right. The main area is split into two panes. The left pane, titled 'Test', contains a code editor with the following SmartApp code:

```

1 /**
2  * Virtual Thermostat
3  *
4  * Author: SmartThings
5  */
6 preferences {
7   section("Choose a temperature sensor..."){
8     input "sensor", "capability.switchLevel", title: "Sensor"
9   }
10  section("Select the heater or air conditioner outlet(s)..."){
11    input "outlets", "capability.switch", title: "Outlets", multiple: true
12  }
13  section("Set the desired temperature..."){
14    input "setpoint", "decimal", title: "Set Temp"
15  }
16  section("When there's been movement from (optional, leave blank to not require motion)..."){
17    input "motion", "capability.motionSensor", title: "Motion", required: false
18  }
19  section("Within this number of minutes..."){
20    input "minutes", "number", title: "Minutes", required: false
21  }
22  section("But never go below (or above if A/C) this value with or without motion..."){
23    input "emergencySetpoint", "decimal", title: "Emer Temp", required: false
24  }
25  section("Select 'heat' for a heater and 'cool' for an air conditioner..."){
26    input "mode", "enum", title: "Heating or cooling?", metadata: [values: ["heat","cool"]]
27  }
28 }
29
30 def installed()
31 {
32   subscribe(sensor, "temperature", temperatureHandler)
33   if (motion) {
34     subscribe(motion, "motion", motionHandler)
35   }
36 }
37
38 def updated()
39 {
40   unsubscribe()
41   subscribe(sensor, "temperature", temperatureHandler)
42   if (motion) {
43     subscribe(motion, "motion", motionHandler)
44   }
45 }
46
47 def temperatureHandler(evt)
48 {
49   def isActive = hasBeenRecentMotion()
50   if (isActive || emergencySetpoint) {
51     evaluate(out.doubleValue, isActive ? setpoint : emergencySetpoint)

```

The right pane, titled 'Location', features a dropdown menu with 'Home' selected and a green 'Set Location' button below it.

104.1 Creating a new SmartApp

To create a new SmartApp, click the *New SmartApp* button from the *My SmartApps* page.

Important: Make sure you have selected the correct Location before creating a new SmartApp. Follow [these steps](#) (page 625) to ensure your code will be published to the correct Location.

There are three different tabs on the *New SmartApp* page that allow you to create a new SmartApp in different ways:

- *From Form* allows you to create a new SmartApp based on the some metadata you can enter into the form.
 - *From Code* allows you to create a new SmartApp directly from existing code. This is useful if you receive the code for a SmartApp - just paste it in to the page and a new SmartApp will be created from it.
 - *From Template* allows you to create a new SmartApp based upon existing SmartApps. This is especially useful if you are new to SmartThings development, since you can start from an existing SmartApp.
-

Important: Only install source code into your account that you fully understand, or that comes from a trusted source.

104.2 Creating a new Device Handler

To create a new Device Handler, click the *Create New Device Handler* button from the *My Device Handlers* page.

Important: Make sure you have selected the correct Location before creating a new Device Handler. Follow [these steps](#) (page 625) to ensure your code will be published to the correct Location.

There are three different tabs on the *New Device Handler* page that allow you to create a new Device Handler in different ways:

- *From Form* allows you to create a new Device Handler based on the some metadata you can enter into the form.
 - *From Code* allows you to create a new Device Handler directly from existing code. This is useful if you receive the code for a Device Handler - just paste it in to the page and a new Device Handler will be created from it.
 - *From Template* allows you to create a new Device Handler based upon existing Device Handlers. This is especially useful if you are new to SmartThings development, since you can start from an existing Device Handlers.
-

Important: Only install source code into your account that you fully understand, or that comes from a trusted source.

104.3 Using the editor

The SmartThings web editor allows you to edit code, and provides syntax highlighting for easy code readability.

You can choose from a variety of themes, key maps, and font sizes to suit your preferences by clicking on the *IDE Settings* button above the editor frame.

Tip: Save often! To avoid losing unsaved changes when your session login to the IDE expires, get in the habit of saving often using *Save* button.

104.4 Using the Simulator

Warning: The simulator may not work reliably at all times, so we recommend that you validate your code on your SmartThings mobile app before deploying it.

The simulator allows you to test your SmartApps or Device Handlers within the IDE, and without requiring you to have the actual physical devices.

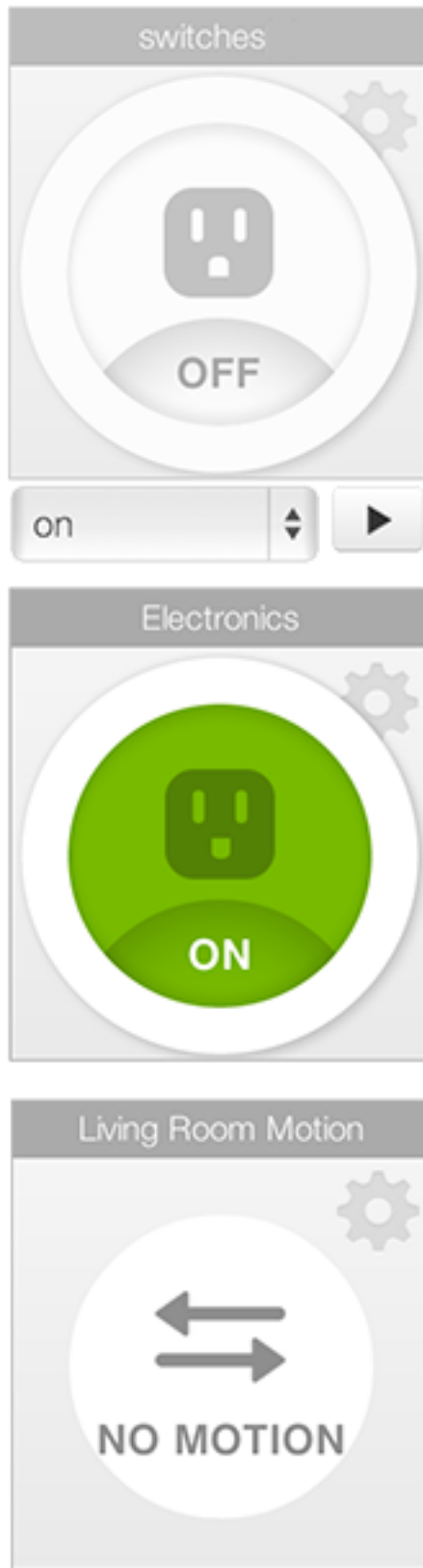
When you run your application in the IDE, it is always running in the simulation framework. The IDE simulator does two very important things to support simulation:

- It acts as a “Virtual Hub” that has virtual devices connected to it.
- It acts as if it was the SmartThings Mobile application to receive and process status updates and support direct user actions on devices through a simulated mobile app control.

The IDE simulation environment also allows you to run the simulator attached to any of the “Locations” defined within your account.

When editing a SmartApp or Device Handler, you can see the simulator on the right of the page. You can choose a Location and click the *Set Location* button, and then input any preferences required by the SmartApp or Device Handler. Click the *Install* button to run the simulator.

When simulating a SmartApp, any selected devices will appear in the IDE, along with controls to actuate the devices:



Logging

SmartApps and Device Handlers can log debugging messages using a built-in logger. This is very useful for debugging purposes.

105.1 Overview

There is an instance of a logger (`log`) injected into each SmartApp and Device Handler available for your use. SmartThings does not currently support a line-by-line, step-through debugger tool; instead, we use logging to debug our custom code. To view the logs, organized by app, click on the *Live Logging* link at the top of the IDE.

105.2 Logging levels

The log instance currently supports these log levels, in decreasing order of severity:

Level	Usage	Description
ER-ROR	<code>log.error(String, Throwable = null)</code>	Runtime errors or unexpected conditions.
WARN	<code>log.warn(String, Throwable = null)</code>	Runtime situations that are unexpected, but not wrong. Can also be used to log use of deprecated APIs.
INFO	<code>log.info(String, Throwable = null)</code>	Interesting runtime events. For example, turning a switch on or off.
DE-BUG	<code>log.debug(String, Throwable = null)</code>	Detailed information about the flow of the SmartApp.
TRACE	<code>log.trace(String, Throwable = null)</code>	Most detailed information.

105.3 Logging exceptions

All log methods accept a second, optional parameter of type `Throwable`. This is useful when catching an exception - you can pass the exception to any of the log methods, and it will include the exception message along with the line number that caused it.

Consider the following example that simply forces a `NullPointerException` by invoking a method on an object that does not exist:

(Real applications should never attempt to handle possible `NullPointerExceptions` like this, of course. It is shown here only to illustrate how to pass the exception to the log methods.)

```
def initialize() {
  try {
    // foo doesn't exist, causing exception
    foo.boom()
  } catch ( ) {
    log.error("caught exception", e)
  }
}
```

Executing the above code would result in the following message in Live Logging:

```
12:42:03 PM: debug caught exception java.lang.NullPointerException: Cannot invoke method boom() on nu
```

105.4 Logging examples

Consider the following simple SmartApp which sets up some switch devices and has an event handler method that will log how many switches are currently turned on.

```
preferences {
  section {
    input "switches", "capability.switch", multiple: true
  }
}

def installed() {
  log.debug "Installed with settings: ${settings}"
  initialize()
}

def updated() {
  log.debug "Updated with settings: ${settings}"
  unsubscribe()
  initialize()
}

def initialize() {
  subscribe(switches, "switch", someEventHandler)
}

def someEventHandler(evt) {
  // returns a list of the values for all switches
  def currSwitches = switches.currentSwitch

  def onSwitches = currSwitches.findAll { switchVal ->
    switchVal == "on" ? true : false
  }

  log.debug "${onSwitches.size()} out of ${switches.size()} switches are on"
}
```

```

3:31:01 PM: trace switch from switches[2] was provided with someEventHandler...creating subscription
3:31:01 PM: trace switch from switches[0] was provided with someEventHandler...creating subscription
3:31:01 PM: trace switch from switches[1] was provided with someEventHandler...creating subscription
3:31:00 PM: trace test is attempting to unsubscribe from all events
3:31:00 PM: debug Updated with settings: [switches:[switches[1], switches[0], switches[2]]]

```

Let's start the above SmartApp execution in the IDE. The first thing that we can see are messages like this:

It is easy to see that the *debug* message came from the `updated()` method.

```

def updated() {
    log.debug "Updated with settings: ${settings}"
    ...
}

```

But where did the other *trace* messages come from? These messages are coming from the SmartApp framework. The SmartApp framework automatically will provide certain information like this during the execution of a SmartApp. Try turning one of the switches on in the IDE. You will see some more of these trace messages coming from the SmartApp framework. You will also see the *debug* message in the `someEventHandler()` method.

```

log.debug "${onSwitches.size()} out of ${switches.size()} switches are on"

```

You should expect to see something like this in live logging.

Note: The newest messages appear at the top of the live logs, not the bottom.

```

3:39:48 PM: debug 2 out of 3 switches are on
3:39:48 PM: trace Replied with 'switch:on'
3:39:48 PM: trace Received command 'on' for device 'Switch Capability switches[1]'

```

Lets see an example of how each one of the log levels look when output to live logging. In the `someEventHandler()` method, I've added the following log messages for this example.

```

log.error "${onSwitches.size()} out of ${switches.size()} switches are on"
log.warn "${onSwitches.size()} out of ${switches.size()} switches are on"
log.info "${onSwitches.size()} out of ${switches.size()} switches are on"
log.debug "${onSwitches.size()} out of ${switches.size()} switches are on"
log.trace "${onSwitches.size()} out of ${switches.size()} switches are on"

```

The output is nice and color coordinated so we can visually see the severity of the various levels.

```

3:56:12 PM: trace 2 out of 3 switches are on
3:56:12 PM: debug 2 out of 3 switches are on
3:56:12 PM: info 2 out of 3 switches are on
3:56:12 PM: warn 2 out of 3 switches are on
3:56:12 PM: error 2 out of 3 switches are on

```

Finally, an example of how the logger can be used in a try/catch block instead of getting the exception.

```
try {
    def x = "some string"
    x.someThingThatDoesNotExist
} catch (all) {
    log.error("Something went horribly wrong!", all)
}
```

GitHub Integration

Warning: Before proceeding to enable GitHub integration in the IDE, be aware that:

1. GitHub IDE integration is not supported outside the US.
2. GitHub IDE integration may negatively impact the performance of the IDE.

As an open platform, we recognize that giving our community developers access to the repository housing our SmartApps and Device Handlers is extremely important. While you can browse the code in the IDE, not having access to the repository itself is limiting. The [SmartThingsCommunity/SmartThingsPublic](#) GitHub repository is now public, allowing you to browse the source code in a more traditional format.

We have also provided an integration with the GitHub repository into the IDE. This will allow SmartThings developers to integrate their forked SmartThingsPublic repository with the IDE, including the ability to make commits to the forked repository using the IDE.

If you just want to browse the source in GitHub, you can do that using the tools you are most comfortable with.

If you want to take advantage of the GitHub integration with the IDE, read on for more information.

Note: A working knowledge of Git and GitHub is assumed in this guide. If you are new to Git and GitHub, we recommend checking out the [GitHub Bootcamp](#) to help you learn the basics. We will walk you through some specific Git steps, but a full discussion/explanation of Git is beyond the scope of this guide.

106.1 Overview

The GitHub IDE integration allows you to integrate your forked SmartThingsPublic repository with the IDE. This allows you to easily view and work with SmartApps or Device Handlers already in the repository, as well as update the versions in your IDE with upstream repository changes, and make commits to your forked repository right from the IDE.

When you setup GitHub integration in the IDE, you will create a fork of the SmartThingsPublic repository in GitHub. This will then be the repository that the IDE will be connected to. When you add files from the repository to the IDE, this is the repository it will look at to get the available files. When you commit changes in the IDE, you are making commits in your remote forked repository.

You will need to manage the syncing of your forked repository with the original SmartThingsPublic repository, just as you would with any forked repository in GitHub.

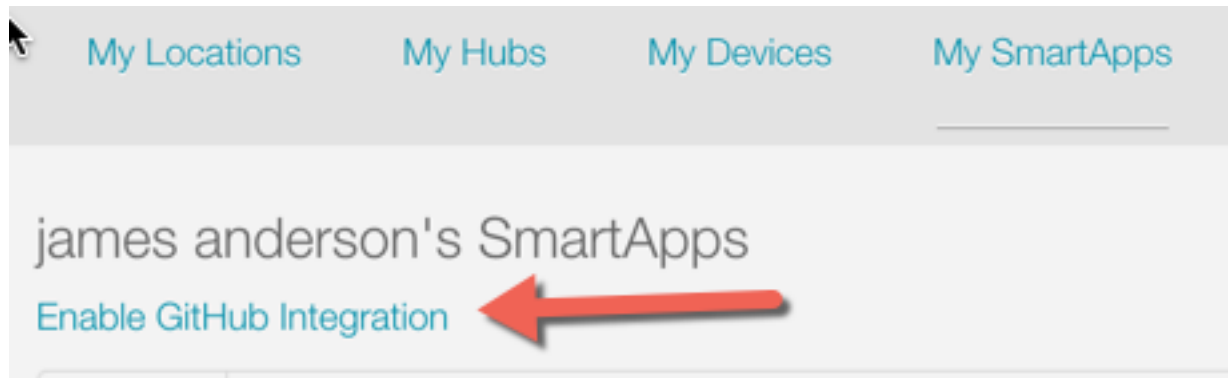
Important: Remember that the IDE is connected to your *remote forked repository in GitHub*. If you create a local clone of your repository, you will need to keep that in sync with the remote repository.

106.2 Setup

To connect your GitHub account with the SmartThingsPublic repository in the IDE, follow these steps.

106.2.1 Step 1 - Enable GitHub integration

Click the *Enable GitHub Integration* link on the *My SmartApps* or *My Device Handlers* page. This will launch a wizard that will guide you through the process.



106.2.2 Step 2 - Connect your GitHub account to SmartThings

On Step 1 of the wizard, follow the instructions to authorize SmartThings to integrate with your GitHub account. Click the *Next* button after you have done this.

Step 1

Connect your GitHub account to SmartThings

Authorize application

SmartThings (localhost) by @SmartThingsCommunity would like permission to access your account

Review permissions



Authorize application

Click the above link and then click *Authorize application* on the GitHub page to connect SmartThings to your GitHub account. This connection allows you to use the IDE to commit changes to and pull down changes from the repositories you add to your SmartThings account. It is also used to create pull requests into the main SmartThingsCommunity repositories when you submit a SmartApp or device handler for publication.

Cancel

Next

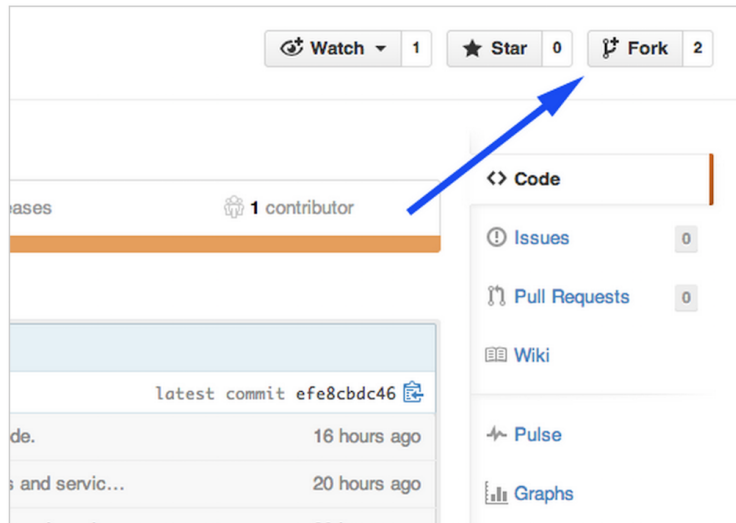
106.2.3 Step 3 - Create a fork

Follow the instructions to fork the SmartThingsCommunity/SmartThingsPublic repository, and then click the *Next* button.

Step 2Fork the *SmartThingsCommunity/SmartThingsPublic* GitHub Repository

Click the above link and then click *Fork* on the GitHub repository page to fork your own copy of the repository. After you've done that and verified that the forked repository has been created, return to this page and click *Next*.

You be able to commit changes made in the IDE into this repository and update SmartApps and device types in the IDE from changes merged into this repository from other sources.


**106.2.4 Step 4 - Clone the forked repository**

Tip: While not required to for submitting changes, this is useful so that you have a local copy of the source code (useful for grepping the source locally, using your favorite editor, etc.), and *is* required to update your fork from the main SmartThingsPublic repository.

Follow these steps to clone your forked repository to your local machine (it is assumed that you have installed and configured Git on your local machine):

On the main page of your forked repository in GitHub, copy the HTTPS clone URL link:

HTTPS clone URL

You can clone with **HTTPS**, **SSH**,
or **Subversion**. 

In a terminal or command prompt, type:

```
git clone <clone URL copied as above>
```

Press Enter. This will create a local clone of your forked repository.

106.2.5 Step 5 - Configure Git to sync fork with SmartThings

If you chose to create a local clone of your forked repository, you should configure it get upstream changes from the original SmartThings repository.

On GitHub, navigate to the SmartThingsCommunity/SmartThingsPublic repository. On the right sidebar of the repository page, copy the clone URL:

Important: This is the clone URL for the main SmartThingsPublic repository, not your fork!



In a terminal or command prompt, change directories to the location of your cloned fork, and type:

```
git remote add upstream <remote URL as copied above>
```

It should look like this:

```
git remote add upstream https://github.com/SmartThingsCommunity/SmartThingsPublic.git
```

Press Enter.

In a terminal or command prompt, type:

```
git remote -v
```

This will show all the configured remotes. You should see an upstream remote configured for the SmartThingsPublic repository.

That's it! You now have connected your GitHub account with the SmartThings IDE. You will now be able to commit changes made in the IDE to this repository, and update SmartApps and Device Handlers in the IDE from changes merged into this repository from other sources.

106.3 Repository structure

The repository is organized by type (SmartApps or Device Handlers) and namespace.

Each SmartApp and Device Handler should be in its own directory, named the same as the SmartApp or Device Handler, and appended with ".src".

For SmartApps:

```
smartapps/<namespace>/<smartapp-name>.src/<smartapp file>.groovy
```

For Device Handlers:

```
devicetypes/<namespace>/<device-type-name>.src/<device handler file>.groovy
```

The namespace is typically your GitHub user name. When you create a SmartApp or Device Handler in the IDE, you provide a namespace, which is then populated in the definition method. This namespace will be used in the directory structure as shown above.

Important: Note that the directory names must all be lowercase and must be consistent with the namespace and the name of the Device Handler or SmartApp. In other words, the directory names must all be lowercase with non-alphanumeric characters replaced with a dash. For example, if a SmartApp has the namespace “My Apps” and the name “My First App” then the path name for it must be `smartapps/my-apps/my-first-app.src/my-first-app.groovy`.

106.4 GitHub integration IDE tour

106.4.1 Color-coded names

The first thing you may notice after enabling GitHub integration is that various SmartApps or Device Handlers are color-coded differently in the IDE. Each name will be color-coded differently depending on its state in the GitHub repository

Hint: Hover your mouse cursor over the name to display a tooltip to give more information.

Black Indicates that the file is unchanged between your forked GitHub repository and the IDE.

Green Indicates that the file is in the IDE only, and not in any repository.

Blue Indicates that the file exists in your GitHub repository, and has been modified in the IDE but not committed to the repository.

Magenta Indicates that the file has been updated in the repository, but not in the IDE. To resolve this, you should click the Update from Repo button, where you will see the file appear in the Obsolete column. More information about the Update from Repo button can be found below.

Red Both the IDE version and repository version have been updated, and are in need of a conflict resolution. To resolve this, you should click the Update from Repo button and follow the steps there (more information about the Update from Repo action can be found below).

Brown Indicates that the SmartApp or Device Handler is unattached to the repository version. Typically this happens when a new SmartApp or Device Handler is created from a template, and the name or namespace hasn't been changed. If you update from the repo without changing the name or namespace, the IDE version will be replaced with the repo version. Typically in this case you would change the name and namespace to be unique for your code.

106.4.2 GitHub actions buttons

When you enable GitHub integration, you will see a few buttons added to the My SmartApps and My DeviceTypes pages in the IDE:



Commit Changes

Clicking the *Commit Changes* button will first prompt you to select what repository you want to commit to, and then launch a wizard that allows you to commit any new or modified code to your forked repository. You can (and should) also add a commit message as you would normally do when making commits in Git.

Update from Repo

Clicking the *Update from Repo* button will first prompt you to select what repository you'd like to update from, and then launch a wizard that allows you to update your IDE code from your forked repository.

The wizard will display three columns, each of which is described below:

Tip: The files considered for this action will depend on if you are on the My SmartApps or My DeviceTypes page in the IDE. Only SmartApps will be considered if launched from My SmartApps, and only device handlers if launched from My DeviceTypes

Obsolete (updated in GitHub) Entries showing in the Obsolete column represent files that you have included in the IDE, but have since been updated in your forked repository (with no conflicts existing). To update your IDE version, select the files you wish to update, and click the Execute Update button.

Conflicted (updated locally and in GitHub) Entries showing in the Conflicted column represent files that have been modified both in the IDE and in your forked repository. To resolve these conflicts, select the files and click the Execute Update button.

New (only in GitHub) Entries showing in the New column are any files found in your forked repository that are not currently in the IDE. To bring these files into your IDE, select the files and click the Execute Update button.

Note: When updating from the repo, you also have the ability to publish any updates (either for yourself or all) by checking the Publish check box.

Settings

This is where you can find information about the repository and branch integrated with the IDE, as well as actions to update, remove, or add new repositories.

106.5 How to

106.5.1 Add files from repository to the IDE

To add files from your forked SmartThingsPublic repository into the IDE, follow these steps:

1. Step 1 - Navigate to the *My SmartApps* or *My Device Handlers* page in the IDE

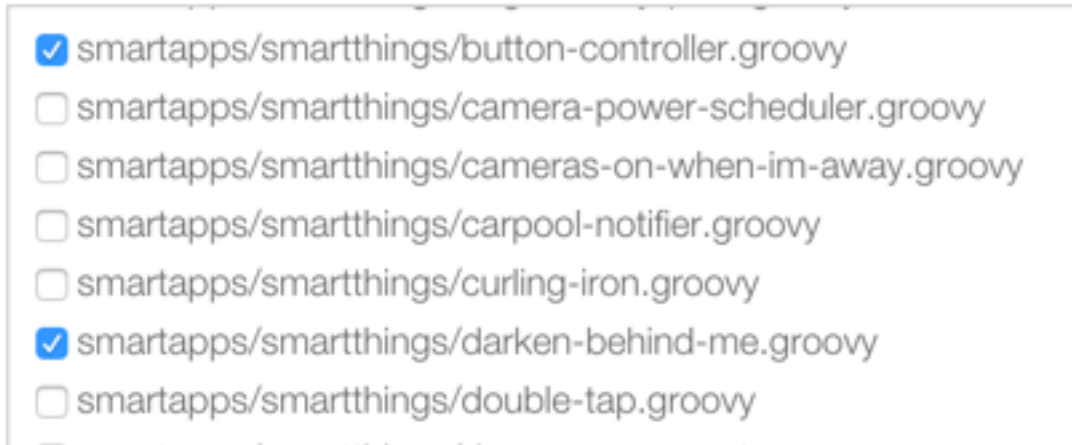
The files available to add to the IDE vary depending upon the context. If you want to add SmartApps to your IDE, navigate to the *My SmartApps* page. If you want to add Device Handlers, navigate to the *My Device Handlers*.

2. Step 2 - Update from Repo

Click the *Update from Repo* button (above the list of SmartApps or device handlers), and select the repo you want to update from.

In the resulting wizard, select the files you want to add to the IDE in the *New (only in GitHub)* column.

New (only in GitHub)



Click the *Execute Update* button in the wizard.

The IDE will now have the files you selected.

106.5.2 Get latest code from SmartThingsPublic repository

Note: To get the latest code from the SmartThingsPublic repository, you need to have cloned your forked repository and configured it to fetch changes from the main (upstream) SmartThingsPublic repository.

See *Step 4 - Clone the forked repository* (page 268) and *Step 5 - Configure Git to sync fork with SmartThings* (page 269) in the *Setup* (page 266) section for more information.

To get the latest code from the SmartThingsPublic repository, follow these steps:

Step 1 - Fetch upstream changes

Open a terminal or command prompt and change directory to the root of your forked repository.

Type `git fetch upstream` and press Enter. This will fetch the branches and their commits from the SmartThingsPublic repository.

Step 2 - Checkout your local master branch

Type `git checkout master` and press Enter.

Step 3 - Merge the changes from upstream/master to your local master branch

Type `git merge upstream/master` and press Enter. This will bring your fork's local master branch up to date with the changes in the SmartThingsPublic master branch.

Step 4 - Push changes to your remote fork

Now that we have our local repository updated synced with the latest SmartThingsPublic repository, we need to push those changes to our remote fork. Remember, this is where the IDE looks for changes (not your local clone!).

Type `git push origin master` and press Enter. This will push all commits in your local repository on the master branch, to the remote (origin) master branch.

Step 5 - Update the IDE version

Now, to update the IDE versions with your updated forked repository, click the *Update from Repo* button on the *My SmartApps* or *My device handlers* page, and select the repo you want to update from.

In the resulting wizard, check the box next to any of the files you want to update in the IDE, and click the *Execute Update* button.

The files you chose to update are now updated in the IDE.

106.5.3 Commit changes in the IDE

To commit changes to a SmartApp or Device Handler, whether it is a new file or already exists in the repository, Click on the *Commit Changes* button on the *My SmartApps* or *My device handlers* and select the repository you want to commit to.

In the resulting wizard, check the box next to the file you want to commit, add a commit message, and press the *Commit Changes* button.

This will make a commit in your fork.

106.5.4 Keep your cloned repo in sync with origin

If you cloned your forked repository to your local machine, you will want to keep it in sync with your remote forked repository in GitHub.

When you make commits in the IDE, you are making a commit and pushing those changes to your forked repository. To sync your cloned repository with the remote forked repository, follow these steps:

Step 1 - Fetch origin changes

Open a terminal or command prompt and change directory to the root of your forked repository.

Type `git fetch origin` and press Enter. This will fetch the branches and their commits from your forked SmartThingsPublic repository.

Step 2 - Checkout your local branch

Type `git checkout master` (substitute `master` for a different branch, if you choose) and press Enter.

Step 3 - Merge the changes from origin/master to your local branch

Type `git merge origin/master` (substitute `master` for a different branch, if you want to merge from a different branch) and press Enter. This will bring your cloned repository's local branch up to date with the changes in your forked SmartThingsPublic branch.

106.6 Best practices

106.6.1 Sync with upstream repository frequently

If you have cloned your forked repository locally, you should merge changes from the upstream SmartThingsPublic repository frequently. This will help prevent your fork from becoming out-of-date with the SmartThingsPublic repository, and minimize the potential for difficult merging of conflicts.

See *Get latest code from SmartThingsPublic repository* (page 272) for instructions on syncing from the upstream SmartThingsPublic repository.

106.7 FAQ

I don't want to grant SmartThings access to my GitHub account. Is there a way around this? Integrating the GitHub repositories with the IDE requires that you grant SmartThings read and write access to your GitHub repositories. If you would rather not grant SmartThings this level of access to your GitHub account, we recommend that you create a new GitHub user to use for SmartThings development. That will allow you to keep your primary GitHub account separate from the SmartThings account.

Do I have to use the GitHub integration? No. The GitHub integration is optional.

Does this change the process for submitting SmartApps or device handlers to SmartThings ? The process for submitting a publication request is essentially the same. The result is slightly different, in that the requests themselves become pull requests in the main SmartThingsPublic repository. This is similar to how it was working previously, but now the pull requests will be visible in the repository since the repository is public.

Can I just make a pull request to the SmartThingsPublic repository, without using the GitHub IDE Integration?

If you make a pull request to the SmartThingsPublic repository, but have not enabled GitHub integration in the IDE, your pull request will not be reviewed or merged in to the SmartThingsPublic repository. Enabling GitHub integration is what allows us to connect your GitHub account with your SmartThings account. If you have enabled the GitHub integration, and then would rather make a pull request to the SmartThingsPublic repository (using the GitHub account you enabled in the IDE) instead of publishing through the IDE, you can. We think it's more efficient to use the tools in the IDE, but nothing prevents you from making a pull request directly in this case.

Where can I find more information about working with Git? See the [Getting help](#) (page 274) section.

I made a commit to my local GitHub fork (not using the IDE), but don't see it when I try to Update from Repo in the IDE.

Did you push your changes to your forked GitHub repository and branch associated with the IDE? Only changes pushed to your forked repository are visible to the IDE - committing changes to your local repository only, without pushing them to the repository and branch associated with the IDE, will not be visible.

I made a commit through the IDE, but I don't see it in my cloned forked repository. Did you merge the latest changes into your local repository? Remember, when you make a commit in the IDE, you are making a commit to your forked version of the SmartThingsPublic repository. If you cloned the repository locally, you need to sync your local repository with the remote repository. See [Keep your cloned repo in sync with origin](#) (page 273) for more information.

I think I found a bug. How do I report it? First, check out the [Getting help](#) (page 274) section below to see if any of the links may answer your questions. If you're confident you've found a bug, and it's not already discussed on the community forums, email support@smarthings.com. For the fastest response, be sure to include your SmartThings user name, your GitHub account name, and specific steps that caused the issue.

106.8 Getting help

Here are some links for getting help working with Git and GitHub:

- [GitHub](#)
- [GitHub Help Page](#)
- [GitHub Bootcamp](#) - useful for getting started with Git.
- [Fork a Repo](#) - documentation on how to fork a repo in GitHub.
- [Sync a Repo](#) - documentation on how to sync a fork to the upstream repository.
- [Pushing to a Remote](#) - documentation on how to push to a remote repository.

If your questions are about the IDE integration, and aren't answered in this documentation, the [SmartThings Community Forums](#) is a great place to leverage the power of our active community developers to help.

Finally, if you have ideas to help improve this documentation, feel free to contact docs@smarththings.com.

Part X

SmartApps

SmartApps are Groovy-based programs that allow a user to tap into the capabilities of their devices to automate their lives.

If you haven't written a SmartApp yet, you should work through the *Writing Your First SmartApp* (page 175).

Anatomy and Life Cycle of a SmartApp

SmartApps are applications that allow users to tap into the capabilities of their devices to automate their lives. Most SmartApps are installed by the user via the SmartThings mobile client application. In addition, a few pre-installed SmartApps are readily available in the SmartThings system out-of-the-box.

107.1 Types of SmartApps

Generally speaking, there are three different kinds of SmartApps: *Event-Handlers*, *Solution Modules*, and *Service Managers*. If you are familiar with back-end web development, then you will be more than capable of developing SmartApps.

107.1.1 Event Handler SmartApps

Event Handler SmartApps are the most common apps developed by our community. They allow you to subscribe to Events from devices and call a handler method upon their firing. This method can then do a variety of things, most commonly invoking a command on another device.

A very simple example of an Event-Handler SmartApp would involve you walking through a door and having the lights turn on automatically.

107.1.2 Solution Module SmartApps

These apps exist within the dashboard of the SmartThings app interface, and are containers for other SmartApps. The idea behind Solution Module SmartApps is to combine SmartApps that, in the real world, intuitively go together. One example of this would be the “Home & Family” section of the dashboard which allows you to see the comings and goings of your family.

107.1.3 Service Manager SmartApps

Service Manager SmartApps are used to connect to LAN or cloud devices, such as a Sonos or a WeMo device. These SmartApps are the connecting glue between the unique protocols of such LAN or cloud devices and a Device Handler you would create for such devices. These Service Manager SmartApps discover LAN or cloud devices and then continue to maintain their connection.

The Service Manager SmartApp must be installed when a user utilizes a device using LAN or the cloud. So, for example, there is a Sonos Service Manager SmartApp that is installed when pairing with a Sonos device.

107.2 SmartApp structure

SmartApps take the form of a single Groovy script. A typical SmartApp script is composed of four sections: *Definition*, *Preferences*, *Predefined Callbacks*, and *Event Handlers*. There is also a *Mappings* section that is required for Cloud-connected SmartApps that will be described later.

```

definition(
  name: "Simple Demo Application",
  namespace: "demo",
  author: "Demo User",
  description: "Turn a light on when a door opens and off when it closes.",
  category: "",
  imageUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
  iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",
  oauth: true)

preferences {
  section("Select devices") {
    input "contact1", "capability.contactSensor", title: "Select contact sensor"
    input "light1", "capability.switch", title: "Select a light"
    input "lock1", "capability.lock", title: "Select a lock"
  }
}

def installed() {
  log.debug "Installed with settings: ${settings}"
  initialize()
}

def updated() {
  log.debug "Updated with settings: ${settings}"
  unsubscribe()
  initialize()
}

def initialize() {
  subscribe contact1, "contact.open", openHandler
  subscribe contact1, "contact.closed", closedHandler
}

def openHandler(evt) {
  light1.on()
  lock1.unlock()
}

def closedHandler(evt) {
  light1.off()
}

```

Metadata that determines how the app is described in the mobile app UI along with other options

Defines what devices and other options are required to install the app. Drive the installation screens in the mobile app UI

Pre-defined methods that are called during SmartApp installation, updating, and deletion

Event handlers specified in event subscriptions and other methods required to implement the SmartApp

107.2.1 Definition

The *definition* section of the SmartApp specifies the name of the app along with other information that identifies and describes it.

107.2.2 Preferences

The *preferences* section is responsible for defining the screens that appear in the mobile app when a SmartApp is installed or updated. These screens allow the user to specify which devices the SmartApp interacts with along with other configuration options that affect its behavior.

107.2.3 Pre-defined callbacks

The following methods, if present, are automatically called at various times during the lifecycle of a SmartApp:

1. `installed()` - Called when a SmartApp is first installed.

2. `updated()` - Called when the preferences of an installed smart app are updated.
3. `uninstalled()` - Called when a SmartApp is uninstalled.
4. `childUninstalled()` - Called for the parent app when a child app is uninstalled (a SmartApp can have child SmartApps).

The `installed()` and `updated()` methods are commonly found in all apps. Since the selected devices may have changed when an app is updated, both of these methods typically set up the same Event subscriptions, so it is common practice to put those calls in an `initialize()` method and call it from both the `installed` and `updated` methods.

The `uninstalled()` method is typically not needed since the system automatically removes subscriptions and schedules when a SmartApp is uninstalled. However, they can be necessary in apps that integrate with other systems and need to perform cleanup on those systems.

107.2.4 Event Handlers

The remainder of the SmartApp contains the event handler methods specified in the Event subscriptions and any other methods necessary for implementing the SmartApp. Event handler methods must have a single argument, which contains the *Event* (page 1017) object.

107.3 SmartApp execution

SmartApps aren't always running. Their various methods are executed when external Events occur. SmartApps execute when any of the following types of Events occur:

1. **Pre-defined callback** - Any of the predefined lifecycle Events described above occur.
 2. **Device state change** - An attribute changes on a device, which creates an Event, which triggers a subscription, which calls a handler method within your SmartApp.
 3. **Location state change** - A location attribute such as *Mode* changes. *Sunrise* and *sunset* are other examples of location events.
 4. **User action on the app** - The user taps a SmartApp icon or shortcut in the mobile app UI.
 5. **Scheduled event** - Using a method like `runIn()`, you call a method within your SmartApp at a particular time.
 6. **Web services call** Using our web services API, you create an endpoint accessible over the web that calls a method within your SmartApp.
-

107.4 Device preferences

The most common type of input in the preferences section specifies what kind of devices a SmartApp works with. For example, to specify that an app requires one contact sensor:

```
input "contact1", "capability.contactSensor"
```

This will generate an input element in the mobile UI that prompts for the selection of a single contact sensor (`capability.contactSensor`). `contact1` is the name of a variable that provides access to the device in the SmartApp.

Device inputs can also prompt for more than one device. So to ask for the selection of one or more switches:

```
input "switch1", "capability.switch", multiple: true
```

You can find more information about SmartApp preferences here.

107.5 Event subscriptions

Subscriptions allow a SmartApp to listen for Events from devices, or from a Location, or from the SmartApp tile in the mobile UI. Device subscriptions are the most common and take the form:

```
subscribe(<device>, "<attribute[.value]>", handlerMethod)
```

For example, to subscribe to all Events from a contact sensor you would write:

```
subscribe(contact1, "contact", contactHandler)
```

The `contactHandler()` method would then be called whenever the sensor opened or closed. You can also subscribe to specific Event values, so to call a handler only when the contact sensor opens write:

```
subscribe(contact1, "contact.open", contactOpenHandler)
```

The `subscribe()` method call accepts either a device or a list of devices, so you don't need to explicitly iterate over each device in a list when you specify `multiple: true` in an input preference.

You can learn more about subscribing to device Events in the [Events and Subscriptions](#) (page 325) section.

107.6 SmartApp sandboxing

SmartApps are developed in a sandboxed environment. The sandbox is a way to limit developers to a specific subset of the Groovy language for performance and security. We have [documented](#) (page 161) the main ways this should affect you.

107.7 Execution location

With the original SmartThings Hub, all SmartApps execute in the SmartThings cloud. With the new Samsung SmartThings Hub, certain SmartApps may run locally on hub or in the SmartThings cloud. Execution location varies depending on a variety of factors, and is managed by the SmartThings internal team.

As a SmartThings developer, you should write your SmartApps to satisfy their specific use cases, regardless of where the app executes. There is currently no way to specify or force a certain execution location.

Preferences and Settings

The preferences section of a SmartApp specifies what kinds of devices and other information is needed in order for the application to run. During the installation of the SmartApp the user is prompted, in the mobile UI, to provide such needed information. The user can present all these inputs on a single page, or break them up into multiple pages.

We strongly recommend you to try out the [web IDE](#) and become familiar with it.

108.1 Preferences overview

Preferences are made up of one or more pages. Later we will see that pages themselves contain one or more sections, which in turn contain one or more elements. The general form of creating preferences looks like:

```
preferences {
  page() {
    section() {
      paragraph "some text"
      input "motionSensors", "capability.motionSensor",
           title: "Motions sensors?", multiple: true
    }
    section() {
      ...
    }
  }
  page() {
    ...
  }
}
```

All inputs from the user are stored in a read-only map called `settings`, and they are available simply by referring to the input name (the first argument to `input()`).

Assuming the following inputs:

```
preferences {
  section() {
    input "someSwitch", "capability.switch"
    input "someText", "text",
    input "someTime", "time"
  }
}
```

The values can be accessed like this:

```
// direct access
log.debug "someSwitch is $someSwitch"
log.debug "someText is $someText"
log.debug "someTime is $someTime"

// via settings
log.debug "settings.someSwitch is $settings.someSwitch"
log.debug "settings.someText is $settings.someText"
log.debug "settings.someTime is $settings.someTime"
```

108.2 Page definition

Pages can be defined two different ways: either by `page(String pageName, String pageTitle) {}` or by `page(options) {}`.

`page(String pageName, String pageTitle) {}`

The code sample below illustrates the first way:

```
preferences {
    // page with name and title
    page("page name", "page title") {
        // sections go here
    }
}
```

`page(options) {}`

This form takes a comma-separated list of name-value arguments.

Note: This is a common Groovy pattern that allows for named arguments to be passed to a method. More info can be found [here](#).

```
preferences {
    page(name: "pageName", title: "page title",
        nextPage: "nameOfSomeOtherPage", uninstall: true) {
        // sections go here
    }
}
```

The valid options are:

name (required) String - Identifier for this page.

title String - The display title of this page.

nextPage String - Used on multi-page preferences only. Should be the name of the page to navigate to next.

install Boolean - Set to `true` to allow the user to install this app from this page. Defaults to `false`. Not necessary for single-page preferences.

uninstall Boolean - Set to `true` to allow the user to uninstall from this page. Defaults to `false`. Not necessary for single-page preferences.

We will see more in-depth examples of pages in the following sections.

108.3 Section definition

Pages can have one or more sections. Think of sections as way to group the inputs you want to gather from the user.

Sections can be created in three different ways:

`section{}`

```
preferences {
    // section with no title
    section {
        // elements go here
    }
}
```

`section(String sectionTitle){}`

```
preferences {
    // section with title
    section("section title") {
        // elements go here
    }
}
```

`section(options, String sectionTitle) {}`

```
preferences {
    // section will not display in IDE
    section(mobileOnly: true, "section title")
}
```

The valid options are:

hideable Boolean - Pass `true` to allow the section to be collapsed. Defaults to `false`.

hidden Boolean - Pass `true` to specify the section is collapsed by default. Used in conjunction with `hideable`. Defaults to `false`.

mobileOnly Boolean - Pass `true` to suppress this section from the IDE simulator. Defaults to `false`.

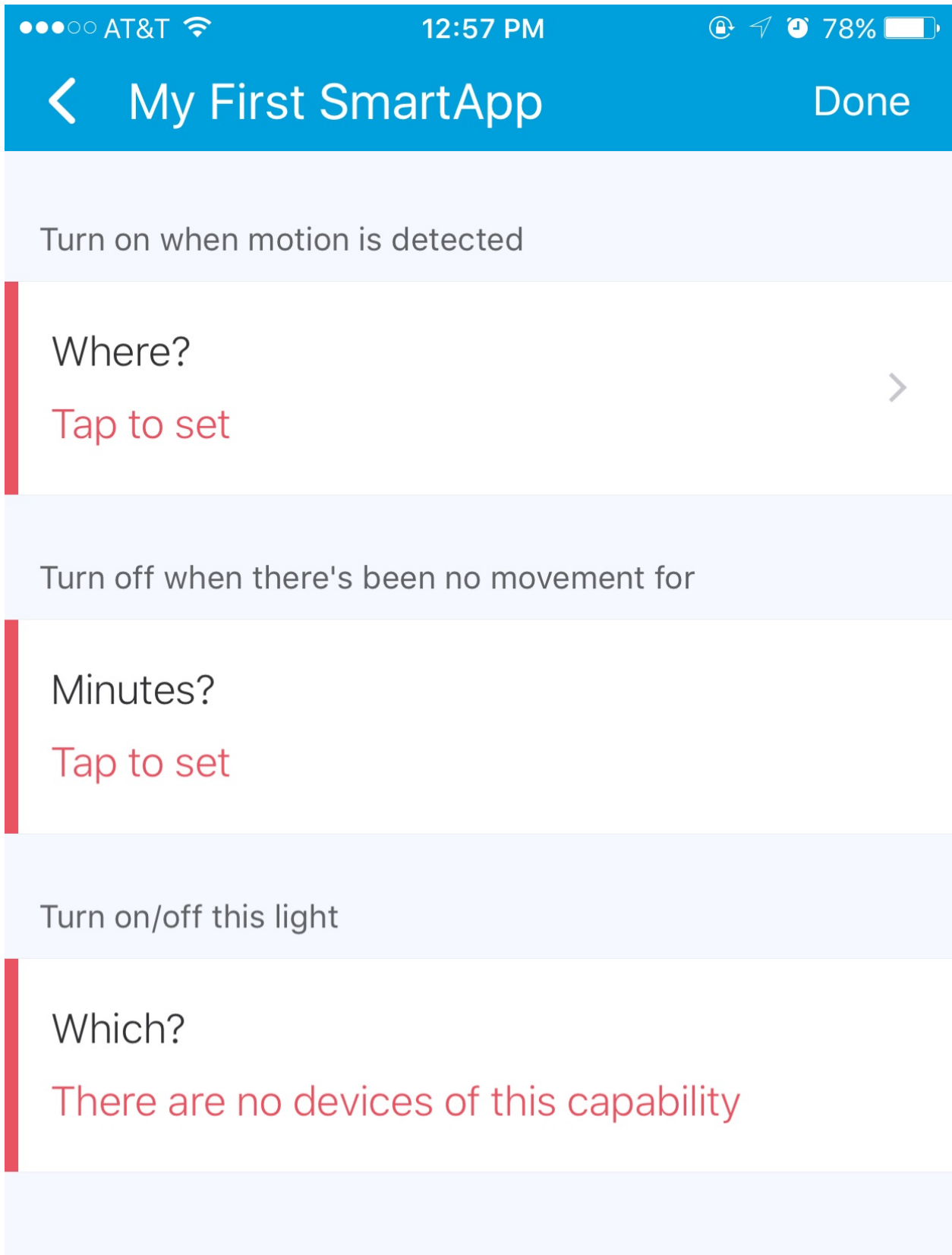
108.4 Single preferences page

A single page preferences declaration is composed of one or more `section` elements, which in turn contain one or more `elements`. Note that there is no `page` defined in the example below. When creating a single-page preferences app, there's no need to define the page explicitly - it's implied. Here's an example:

```
preferences {
    section("Turn on when motion is detected") {
        input "themotion", "capability.motionSensor", required: true, multiple: true, title: "When motion is detected"
    }
    section("Turn off when there's been no movement for") {
        input "minutes", "number", required: true, title: "Minutes?"
    }
    section("Turn on/off this light") {
        input "theswitch", "capability.switch", required: true
    }
}
```

```
}
```

Which would be rendered in the mobile app UI as:



Assign a name
Tap to set

Note that in the above example, we did not specify the *name* or *mode* input in the `preferences` section of the code, yet they appeared in the UI of our mobile app at the bottom (“Assign a name” and “Set for specific mode(s)”). When defining single-page preferences, name and mode are automatically added. Also note that inputs that are marked as `required: true` are displayed prominently in red color by the mobile app, so that the user knows they are required. The mobile application will prevent the user from going to the next page or installing the SmartApp without entering required inputs.

108.5 Multiple preferences pages

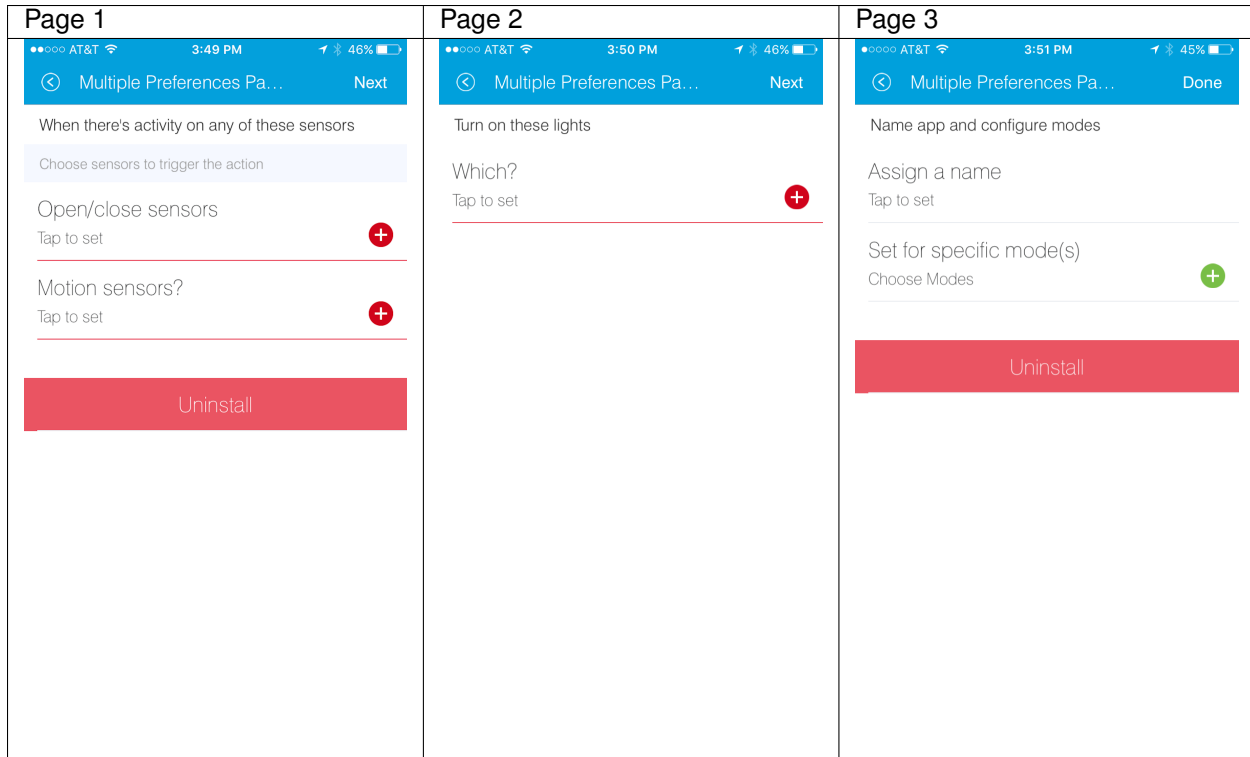
Preferences can also be broken up into multiple pages. Each page must contain one or more *section* elements. Each page specifies a *name* property that is referenced by the *nextPage* property. The *nextPage* property is used to define the flow of the pages.

Note: Unlike single page preferences, the name and mode control fields are not automatically added, and must be specified on the desired page or pages.

Here’s an example that defines three pages:

```
preferences {
  page(name: "pageOne", title: "When there's activity on any of these sensors", nextPage: "pageTwo") {
    section("Choose sensors to trigger the action") {
      input "contactSensors", "capability.contactSensor",
           title: "Open/close sensors", multiple: true
      input "motionSensors", "capability.motionSensor",
           title: "Motion sensors?", multiple: true
    }
  }
  page(name: "pageTwo", title: "Turn on these lights", nextPage: "pageThree") {
    section {
      input "switches", "capability.switch", multiple: true
    }
  }
  page(name: "pageThree", title: "Name app and configure modes", install: true, uninstall: true) {
    section([mobileOnly:true]) {
      label title: "Assign a name", required: false
      mode title: "Set for specific mode(s)", required: false
    }
  }
}
```

The resulting pages in the mobile app would show the name and mode control fields only on the third page, and the uninstall button on the first and third pages:



108.6 Preference elements and inputs

Preference pages (single or multiple) are composed of one or more *sections*. Each *section*, in turn, contains one or more of the following elements:

108.6.1 paragraph

Text that is displayed on the page for messaging and instructional purposes.

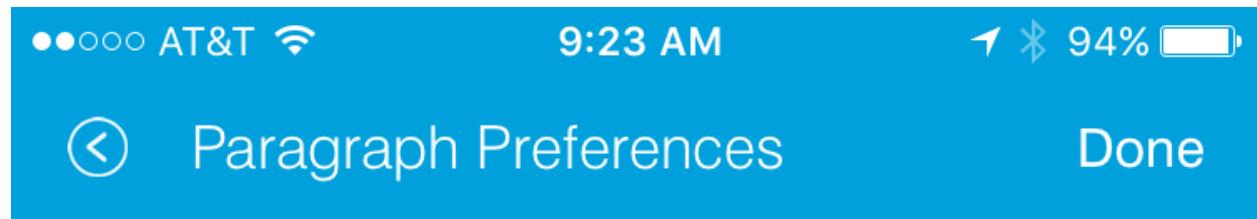
Example:

```

preferences {
    section("paragraph") {
        paragraph "This is how you can make a paragraph element"
        paragraph image: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
            title: "paragraph title",
            required: true,
            "This is a long description that rambles on and on and on..."
    }
}

```

The above `preferences` definition would render the mobile app UI as:



paragraph

This us how you can make a paragraph element



paragraph title

This is a long description that keeps rambles on and on and on...

Assign a name

Tap to set

Set for specific mode(s)

Choose Modes



Uninstall

Valid options are:

title String - The title of the paragraph.

image String - URL of image to use, if desired.

required Boolean - `true` or `false` to specify this input is required. Defaults to `false`.

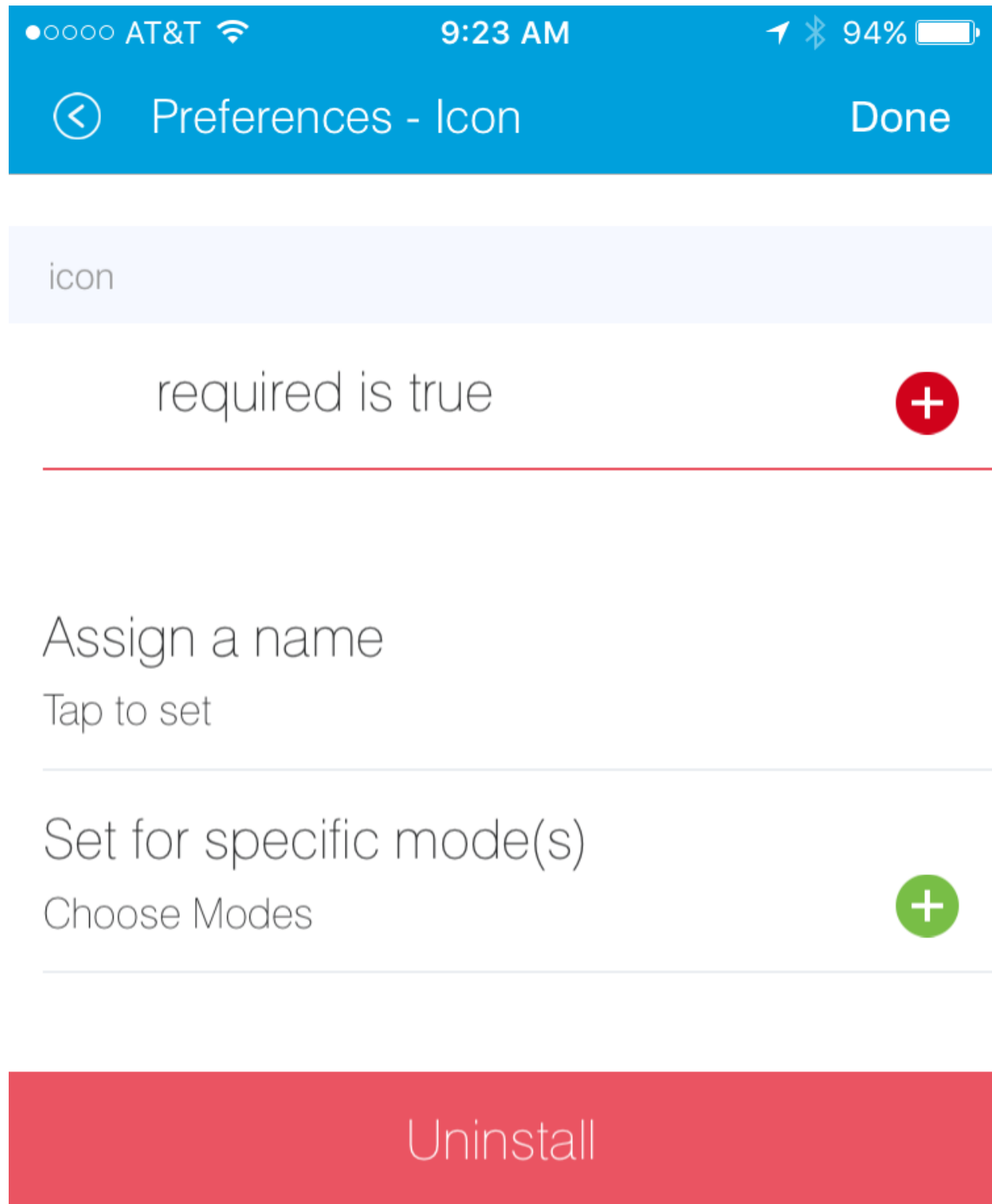
108.6.2 icon

Allows the user to select an icon to be used when displaying the app in the mobile UI.

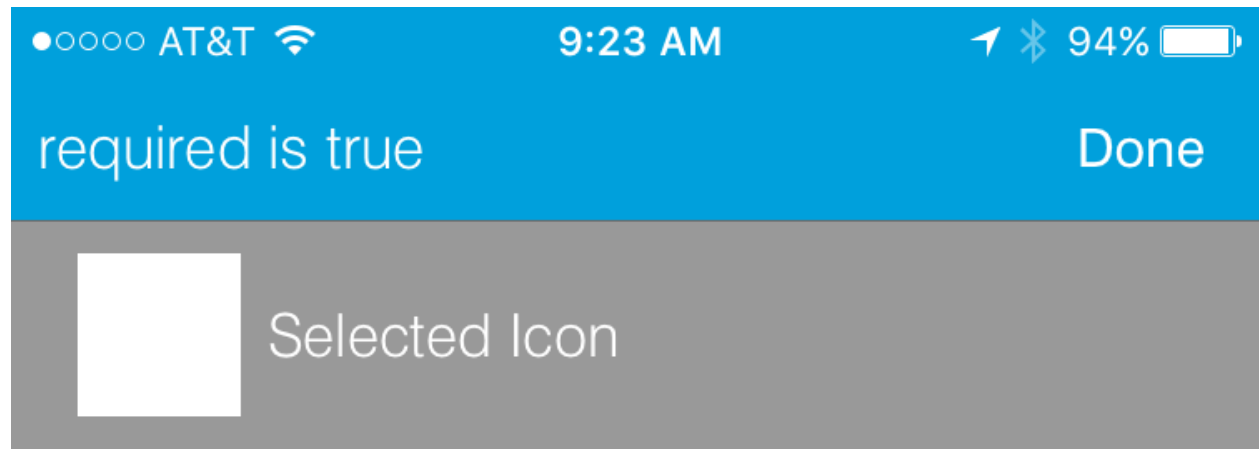
Example:

```
preferences {
  section("paragraph") {
    icon(title: "required is true",
         required: true)
  }
}
```

The above preferences definition would render the mobile app UI as:



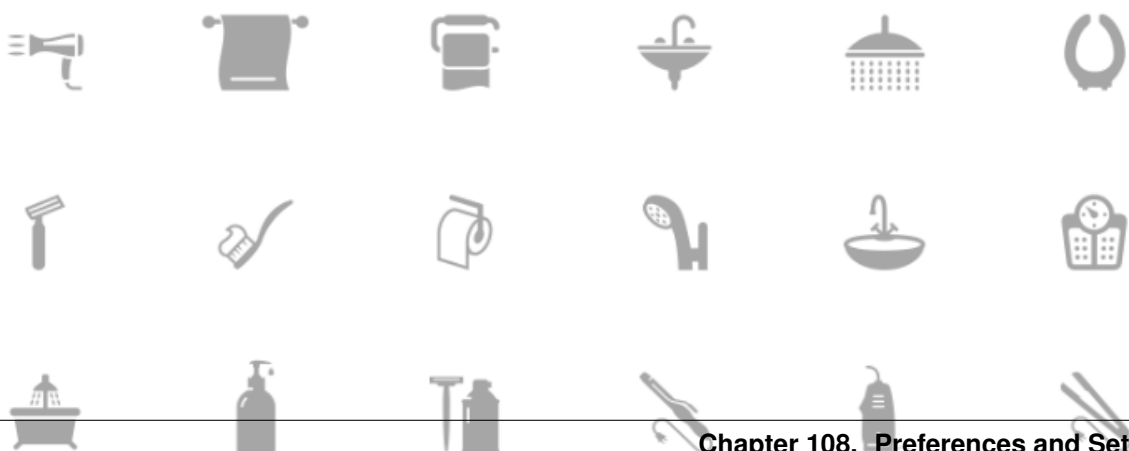
Tapping the *icon* UI element would then allow the user to choose an icon:



Appliances



Bath



Valid options are:

title String - The title of the icon.

required Boolean - true or false to specify this input is required. Defaults to false.

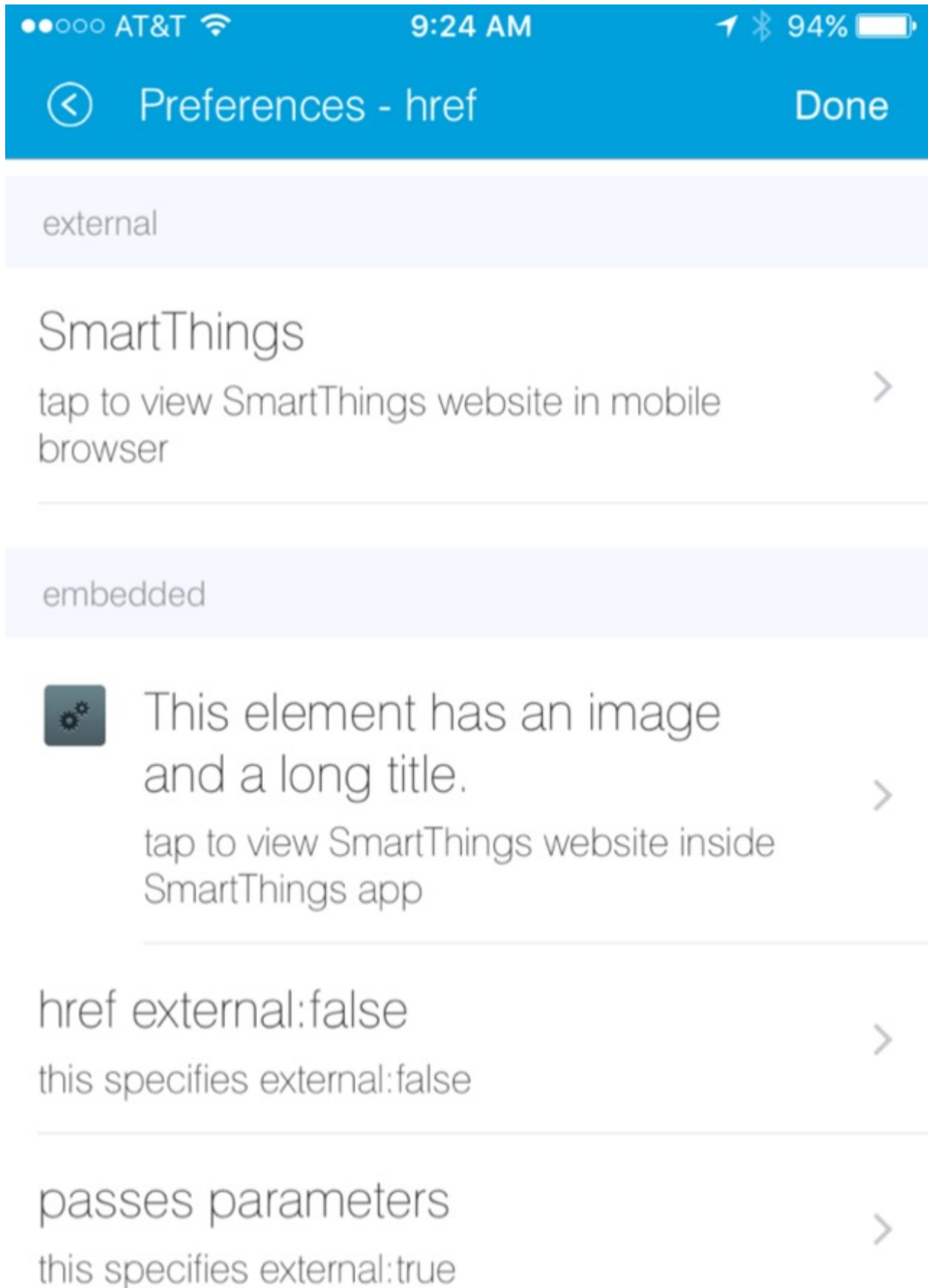
108.6.3 href

A control that selects an external HTML page or another preference page.

Example of using *href* to visit a URL:

```
preferences {
  section("external") {
    href(name: "hrefNotRequired",
         title: "SmartThings",
         required: false,
         style: "external",
         url: "http://smarththings.com/",
         description: "tap to view SmartThings website in mobile browser")
  }
  section("embedded") {
    href(name: "hrefWithImage", title: "This element has an image and a long title.",
         description: "tap to view SmartThings website inside SmartThings app",
         required: false,
         image: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
         url: "http://smarththings.com/")
  }
}
```

The above preferences would render the mobile app UI as:



Example of using *href* to link to another preference page (dynamic pages are discussed later in this section):

```
preferences {
    page(name: "hrefPage")
    page(name: "deadEnd")
}

def hrefPage() {
    dynamicPage(name: "hrefPage", title: "href example page", uninstall: true) {
        section("page") {
            href(name: "href",
                title: "dead end page",
                required: false,
                page: "deadEnd")
        }
    }
}

def deadEnd() {
    dynamicPage(name: "deadEnd", title: "dead end page") {
        section("dead end") {
            paragraph "this is a simple paragraph element."
        }
    }
}
```

You can use the *params* option to pass data to dynamic pages:

```
preferences {
    page(name: "firstPage")
    page(name: "secondPage")
}

def firstPage() {
    def hrefParams = [
        foo: "bar",
        someKey: "someVal"
    ]

    dynamicPage(name: "firstPage", uninstall: true) {
        section {
            href(name: "toSecondPage",
                page: "secondPage",
                params: hrefParams,
                description: "includes params: ${hrefParams}")
        }
    }
}

// page def must include a parameter for the params map!
def secondPage(params) {
    log.debug "params: ${params}"
    dynamicPage(name: "secondPage", uninstall: true, install: true) {
        section {
            paragraph "params.foo = ${params?.foo}"
        }
    }
}
```

Valid options are:

title String - the title of the element.

required Boolean - `true` or `false` to specify this input is required. Defaults to `false`.

description String - the secondary text of the element

external (deprecated - use style instead) Boolean - `true` to open URL in mobile browser application, `false` to open URL within the SmartThings app. Defaults to `false`.

style String - Controls how the link will be handled. Specify “external” to launch the link in the mobile device’s browser. Specify “embedded” to launch the link within the SmartThings mobile application. Specify “page” to indicate this is a preferences page.

If `style` is not specified, but `page` is, then `style: "page"` is assumed. If `style` is not specified, but `url` is, then `style: "embedded"` is assumed.

Currently, Android does not support the “external” style option.

url String - The URL of the page to visit. You can use query parameters to pass additional information to the URL (for example, `http://someurl.com?param1=value1¶m2=value1`).

params Map - Use this to pass parameters to other preference pages. If doing this, make sure your page definition method accepts a single parameter (that will be this `params` map). See the `page-params-by-href` example at the end of this document for more information.

page String - Used to link to another preferences page. Not compatible with the `external` option.

image String - URL of an image to use, if desired.

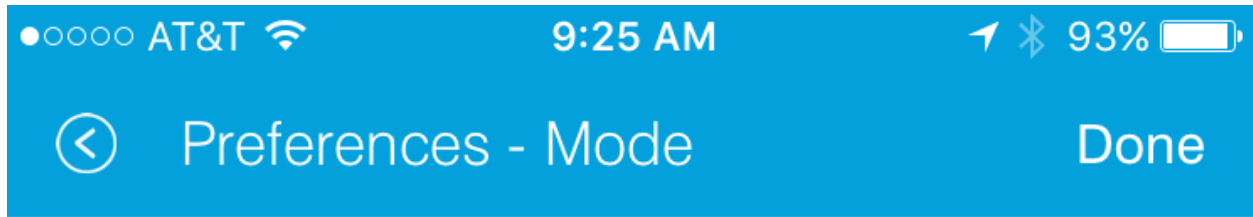
108.6.4 mode

Allows the user to select which modes the app executes in. Automatically generated by single-page preferences.

Example:

```
preferences {
    page(name: "pageOne", title: "page one", nextPage: "pageTwo", uninstall: true) {
        section("section one") {
            paragraph "just some text"
        }
    }
    page(name: "pageTwo", title: "page two") {
        section("page two section one") {
            mode(name: "modeMultiple",
                title: "pick some modes",
                required: false)
            mode(name: "modeWithImage",
                title: "This element has an image and a long title.",
                required: false,
                multiple: false,
                image: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png")
        }
    }
}
```

The second page of the above example would render in the mobile UI as:



page two

page two section one

pick some modes

Choose Modes



This element has an image
and a long title.

Choose Modes



Valid options are:

title String - the title of the mode field.

required Boolean - `true` or `false` to specify this input is required. Defaults to `false`.

multiple Boolean - `true` or `false` to specify this input allows selection of multiple values. Defaults to `true`.

image String - URL of an image to use, if desired.

Note: There are a couple of different ways to use modes that are worth pointing out. The first way is to use modes as a type of enum input like this:

```
input "modes", "mode", title: "only when mode is", multiple: true, required: false
```

This method will automatically list the defined modes as the options. Please note when using modes in this way that the modes are just data and can be accessed in the SmartApp as such. This does not effect SmartApp execution. In this scenario, it is up to the SmartApp itself to react to the mode changes.

The second example actually controls whether the app is executed based on the modes selected:

```
mode( title: "set for specific mode(s)")
```

Both of these methods of using modes are valid. The impact on SmartApp execution is different in each scenario and it is up to the SmartApp developer to properly label whichever form is used and code the app accordingly.

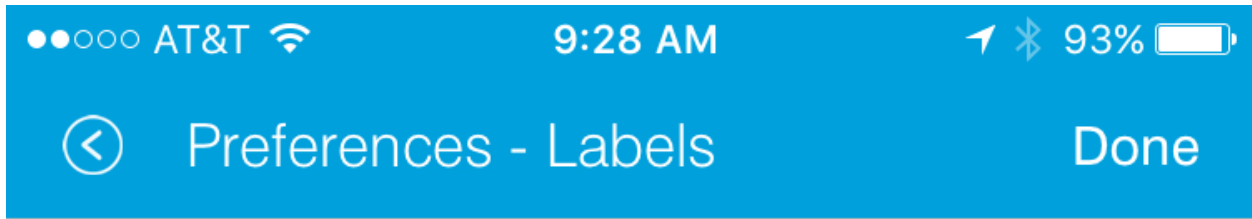
108.6.5 label

Allows the user to name the app installation. Automatically generated by single-page preferences.

Example:

```
preferences {
    section("labels") {
        label(name: "label",
            title: "required:false",
            required: false,
            multiple: false)
        label(name: "labelRequired",
            title: "required:true",
            required: true,
            multiple: false)
        label(name: "labelWithImage",
            title: "This element has an image and a title.",
            description: "image and a title",
            required: false,
            image: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png")
    }
}
```

The above preferences definition would render in the mobile UI as:



labels

required:false

Tap to set

required:true

Tap to set



This element has an image and a title.

image and a title

Assign a name

Tap to set

Set for specific mode(s)

Note: Images do not currently render in `label` inputs on Android.

Valid options are:

title String - the title of the label field.

description String - the text in the input field.

required Boolean - `true` or `false` to specify this input is required. Defaults to `false`. Defaults to `true`.

image String - URL to an image to use, if desired.

108.6.6 app

Provides user-initiated installation of child apps.

108.6.7 input

Allows the user to select devices or enter values to be used during execution of the SmartApp.

Inputs are the most commonly used preference elements. They can be used to prompt the user to select devices that provide a certain capability, or devices of a specific type, or constants of various kinds.

Input element method calls take two forms.

The “shorthand” form passes in the name and type unnamed as the required first two parameters, and any other arguments as named options:

```
preferences {
    section("section title") {
        // name is "temperature1", type is "number"
        input "temperature1", "number", title: "Temperature"
    }
}
```

The second form explicitly specifies the name of each argument:

```
preferences {
    section("section title") {
        input(name: "color", type: "enum", title: "Color", options: ["Red", "Green", "Blue", "Yellow"])
    }
}
```

Valid input options are:

capitalization (Note - this feature is currently only supported on iOS devices) String - if the input is a text field, this controls the behavior of the auto-capitalization on the mobile device. `none` specifies to not enable auto-capitalization for any word. `sentences` will capitlize the first letter of each sentence. `all` will use all caps. `words` will capitalize every word. The default is `words`.

defaultValue Object - if specified, a default value for this input.

name String - name of variable that will be created in this SmartApp to reference this input.

title String - title text of this element.

description String - default value of the input element.

multiple Boolean - `true` or `false` to specify this input allows selection of multiple devices of the input type (if you have more than one). Defaults to `true`. For example, in the motion sensor example above, setting this to `true` will allow you to select more than one motion sensor, provided you have more than one.

range A range for numeric (number and decimal) that restricts the valid entries to values within the range. For example, `range: "2..7"` will only allow inputs between 2 and 7 (inclusive). `range: "-5..8"` allows inputs between -5 and 8. A value of "*" will allow any numeric value on that side of the range. Use `range: "*..*"` to allow the user to enter any value, negative or positive. Note that without specifying a range that allows negative numbers, the mobile clients will only show a keypad to allow positive numeric entries.

required Boolean - `true` to require the selection of a device for this input or `false` to not require selection.

submitOnChange Boolean - `true` to force a page refresh after input selection or `false` to not refresh the page. This is useful when creating a dynamic input page.

options List - used in conjunction with the enum input type to specify the values the user can choose from. Example: `options: ["choice 1", "choice 2", "choice 3"]`.

type String - one of the names from the following table:

Name	Comment
<code>capability.capabilityName</code>	Prompts for all the devices that match the specified capability. See the <i>Preferences Reference</i> column of the <i>Capabilities Reference</i> (page 655) table for possible values.
<code>device.deviceTypeName</code>	Prompts for all devices of the specified type. See <i>Using device-specific inputs</i> (page 306) for more information.
<code>bool</code>	A <code>true</code> or <code>false</code> value (value returned as a boolean).
<code>boolean</code>	A <code>"true"</code> or <code>"false"</code> value (value returned as a string). It's recommended that you use the <code>"bool"</code> input instead, since the simulator and mobile support for this type may not be consistent, and using <code>"bool"</code> will return you a boolean (instead of a string). The <code>"boolean"</code> input type may be removed in the near future.
<code>decimal</code>	A floating point number, i.e. one that can contain a decimal point
<code>email</code>	An email address
<code>enum</code>	One of a set of possible values. Use the <i>options</i> element to define the possible values.
<code>hub</code>	Prompts for the selection of a hub
<code>icon</code>	Prompts for the selection of an icon image
<code>number</code>	An integer number, i.e. one without decimal point
<code>password</code>	A password string. The value is obscured in the UI and encrypted before storage
<code>phone</code>	A phone number
<code>time</code>	A time of day. The value will be stored as a string in the Java <code>SimpleDateFormat</code> (e.g., <code>"2015-01-09T15:50:32.000-0600"</code>)
<code>text</code>	A text value

108.6.8 Using device-specific inputs

If a specific device is required for a SmartApp, the device itself can be used instead of the capability, with the "device.<deviceName>" input type. For example, if your SmartApp specifically requires a device named "My Fancy Device", you can prompt the user for that device this way:

```
input "myDevice", "device.myFancyDevice"
```

The format of the device name is determined by the following algorithm:

1. Remove "device." prefix ("device.myFancyDevice" -> "myFancyDevice").
2. Capitalize the result ("myFancyDevice" -> "MyFancyDevice").
3. Split the result by camel case ("MyFancyDevice" -> ["My", "Fancy", "Device"]).
4. Join result with a space (["My", "Fancy", "Device"] -> "My Fancy Device").
5. Replace occurrences of any of these strings in the result with the following, as shown:

Original	Replaced With
"Smart Sense"	"SmartSense"
"Smart Power Outlet V 1"	"SmartPower Outlet V1"
"Smart Power Outlet"	"SmartPower Outlet"
"Open Closed Sensor"	"Open/Closed Sensor"
"On Off"	"On/Off"
"Door Window"	"Door/Window"
"Motion Temp Sensor"	"Motion/Temp Sensor"
"Z Wave"	"Z-Wave"
"Zwave"	"Z-Wave"
"Smart Phone"	"Mobile Presence"
"Mobile Presence"	"Mobile Presence"

Here are a few examples:

Device Preference Input	Device Name Searched For
"device.myFancyDevice"	"My Fancy Device"
"device.ecobeeThermostat"	"Ecobee Thermostat"
"device.myOnOffDevice"	"My On/Off Device"

When using device.<name> inputs, the platform first looks up which Device Handler it is, then finds any devices of that type for that Location. The algorithm searches for a Device Handler in the following order:

1. A Device Handler published by SmartThings that matches the name.
2. A Device Handler published by the current user that matches the name.

If there are multiple Device Handlers with the same name, the first Device Handler found will be returned. Only the name of the Device Handler is searched for; namespace is not considered.

There are some caveats to be aware of due to the way the algorithm works:

- The name of the device should have every word capitalized.
- Use of numbers can cause unexpected results.
- Use of spaces in the device input can cause unexpected results.

Here are some examples that illustrate this:

Device Preference Input	Device Name Searched For
"device.myDevice v1"	"My Device v 1"
"device.myDeviceV1"	"My Device V 1"
"device.myDevicev1"	"My Devicev 1"
"device.mydevice"	"Mydevice"

108.7 Hide when empty

Inputs, sections, and pages support the `hideWhenEmpty` attribute. This attribute will hide the element that it is associated with when the element is empty. For example, if you have an input that prompts the user for an audio device, but that user does not have any audio devices, the `hideWhenEmpty` attribute will hide the input from the user. Let's take a look at a few examples.

Add the `hideWhenEmpty` attribute to SmartApp inputs to completely hide UI control if there are no devices available. In this example, the SmartApp will not display the valve input if there were no valves in this user's Location, but would display the switch input even if there were no switches.

```
preferences {
    section {
        input "switches", "capability.switch", title: "Select a switch"
        input "valves", "capability.valve", title: "Select a valve", hideWhenEmpty: true, required: true
    }
}
```

Adding the `hideWhenEmpty` attribute to a *section* or a *page* will cascade the attribute down to all of the child *input* elements. This means that adding the `hideWhenEmpty` attribute to any parent element is in effect the same as adding the `hideWhenEmpty` attribute to all of the child *input* elements. Let's look at a few examples.

The following example will hide the entire section if there are no valves and no switches. If the user did have a switch or a valve, then the section would be displayed with only the input element that is available.

```
preferences {
    section(hideWhenEmpty: true) {
        input "switches", "capability.switch", title: "Select a switch"
        input "valves", "capability.valve", title: "Select a valve", required: false
    }
}
```

The last example illustrates how this attribute applies to an entire page element. In this case, any section will be hidden if all of its input elements are absent. For example, if the switch device is available but the valve device is not available, then the section with switches and valves will still display. However, if both switch and valve devices are entirely absent, then the section with switches and valves will not display.

```
preferences {
    page(name: "mainPage", title: "Select some things", hideWhenEmpty: true) {
        section {
            input "switches", "capability.switch", title: "Select a switch"
            input "valves", "capability.valve", title: "Select a valve", required: false
        }
        section {
            input "audio", "capability.musicPlayer", title: "Select a music player"
        }
    }
}
```

It is worth noting that in the last example, the audio input does not have the usual `required: false` attribute. This is because the input will not be displayed if there are no audio devices associated to this Location. However, the SmartApp would have to be able to handle a `null` value for that input. Also, it is worth remembering that if the user does have an audio device in this Location, the default value of `required: true` will be applicable.

108.7.1 Working with other input types

We've seen how the `hideWhenEmpty` attribute works with device inputs, but what about other types of inputs like Number, text, or Boolean inputs?

These types of inputs will always appear because they can never have empty selections. It is possible to hide these kinds of input elements if they relate to another input element. Let's look at an example where we have two inputs, an audio device input, and a volume input. The volume input can never be empty, so we can't hide it. But it is related to the audio input which can be empty and hidden. In this case, we can hide the entire section containing the two inputs by telling the volume input to hide if the audio input is empty. We do this by referencing the name of the related input.

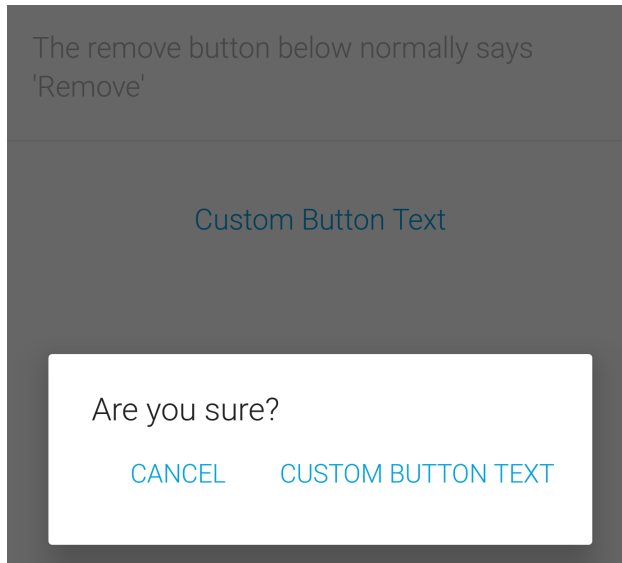
```
preferences {
    page(name: "mainPage", title: "Select some things", hideWhenEmpty: true) {
        section {
            input "audio", "capability.musicPlayer", title: "Select a music player"
            input "volume", "number", title: "Set it to this volume level", hideWhenEmpty: "audio"
        }
    }
}
```

108.8 Custom Remove button

By default, a *Remove* button is added to the bottom of a preferences page that specifies `uninstall: true`. This button can be customized by using the `remove()` method:

```
page(name: "firstPage") {
    section {
        paragraph "The remove button below normally says 'Remove'"
    }
    remove("Custom Button Text")
}
```

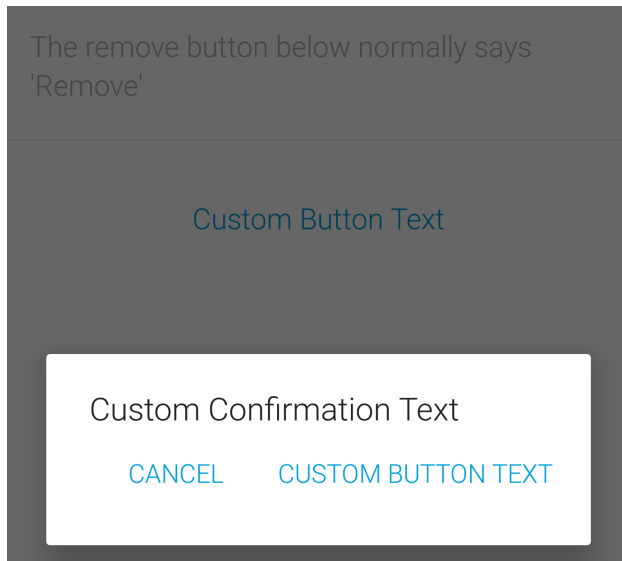
The specified text is used as the label of the button on the page, as well as the label of the confirmation button on the resulting confirmation dialog:



We can also specify custom confirmation text:

```
page(name: "firstPage") {
  section {
    paragraph "The remove button below normally says 'Remove'"
  }
  remove("Custom Button Text", "Custom Confirmation Text")
}
```

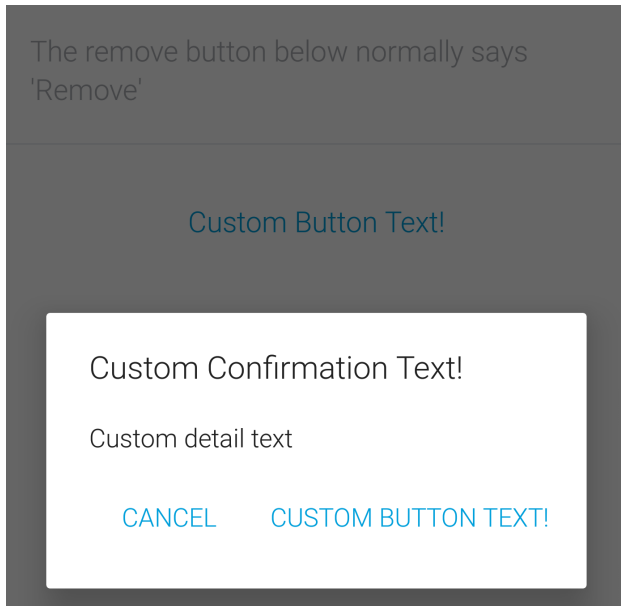
This renders in the mobile UI as:



Finally, we can specify custom detail text to show on the confirmation dialog:

```
page(name: "firstPage") {
  section {
    paragraph "The remove button below normally says 'Remove'"
  }
  remove("Custom Button Text!", "Custom Confirmation Text!", "Custom detail text")
}
```

This renders in the mobile UI as:



The use of `remove ()` must follow these rules:

- It must be defined after all other sections.
- It must not be nested inside a section.
- It can only be used inside a page.
- It must only be used once per page.

If these rules are not followed, exceptions are thrown and error messages are displayed when pressing *Save*.

`remove ()` also sets the page `uninstall` to `true`.

108.9 Dynamic preferences

One of the most powerful features of multi-page preferences is the ability to dynamically generate the content of a page based on previous selections or external inputs, such as the data elements returned from a web services call. The following example shows how to create a two-page preferences SmartApp where the content of the second page depends on the selections made on the first page.

```
preferences {
    page(name: "page1", title: "Select sensor and actuator types", nextPage: "page2", uninstall: true)
    section {
        input("sensorType", "enum", options: [
            "contactSensor": "Open/Closed Sensor",
            "motionSensor": "Motion Sensor",
            "switch": "Switch",
            "moistureSensor": "Moisture Sensor"])

        input("actuatorType", "enum", options: [
            "switch": "Light or Switch",
            "lock": "Lock"]
    )
}
```

```

    }
  }
  page(name: "page2", title: "Select devices and action", install: true, uninstall: true)
}

def page2() {
  dynamicPage(name: "page2") {
    section {
      input(name: "sensor", type: "capability.$sensorType", title: "If the $sensorType device")
      input(name: "sensorAction", type: "enum", title: "is", options: attributeValues(sensorType))
    }
    section {
      input(name: "actuator", type: "capability.$actuatorType", title: "Set the $actuatorType")
      input(name: "actuatorAction", type: "enum", title: "to", options: actions(actuatorType))
    }
  }
}

private attributeValues(attributeName) {
  switch(attributeName) {
    case "switch":
      return ["on", "off"]
    case "contactSensor":
      return ["open", "closed"]
    case "motionSensor":
      return ["active", "inactive"]
    case "moistureSensor":
      return ["wet", "dry"]
    default:
      return ["UNDEFINED"]
  }
}

private actions(attributeName) {
  switch(attributeName) {
    case "switch":
      return ["on", "off"]
    case "lock":
      return ["lock", "unlock"]
    default:
      return ["UNDEFINED"]
  }
}

```

The previous example shows how you can achieve dynamic behavior between pages. Next, with the `submitOnChange` input attribute you can also have dynamic behavior in a single page.

```

preferences {
  page(name: "examplePage")
}

def examplePage() {
  dynamicPage(name: "examplePage", title: "", install: true, uninstall: true) {
    section {

```

```
        input(name: "dimmers", type: "capability.switchLevel", title: "Dimmers",
              description: null, multiple: true, required: false, submitOnChange: true)
    }

    if (dimmers) {
        // Do something here like update a message on the screen,
        // or introduce more inputs. submitOnChange will refresh
        // the page and allow the user to see the changes immediately.
        // For example, you could prompt for the level of the dimmers
        // if dimmers have been selected:

        section {
            input(name: "dimmerLevel", type: "number", title: "Level to dim lights to...", required: true)
        }
    }
}
```

Note: When a submitOnChange input is changed, the whole page will be saved and then a refresh is triggered with the saved page state. This means that all of the methods will execute each time you change a submitOnChange input.

108.9.1 dynamicPage() options

Any valid option for `page()` will work for `dynamicPage()` also. In addition, the `refreshInterval` input option is specific to `dynamicPage()` method:

```
preferences {
    page(name: "page0")
    page(name: "page1")
    page(name: "page3")
}

...

def page1() {
    dynamicPage(name: "page1", title: "Page 1", nextPage: "page2", refreshInterval: 5, minInstall: "108.9.1")
}
```

refreshInterval Integer - refreshes the specific page of the SmartApp on the mobile device for the integer number of seconds. In the above example, it refreshes the `page1` every 5 seconds.

108.10 Private settings

Some SmartApps may need to reference sensitive data, such as API keys or secrets. These should not be placed directly in the source code, since anyone with access to the source will then be able to view this sensitive information.

Instead, you should specify `appSettings` in the SmartApp's definition:

```
definition(  
    name: "your app name",  
    namespace: "your-namespace",  
    // ...  
) {  
    appSetting "setting1"  
    appSetting "setting2"  
}
```

The string passed to `appSetting` will be the name of the setting. The actual values are set on the Edit SmartApp page, accessed by pressing the *App Settings* button. Scroll down the page, expand the *Settings* group, and set the values as needed.

The values are stored in a map in `app.appSettings`. You can access the values like this:

```
definition(  
    //...  
) {  
    appSetting "apiSecret"  
}  
  
// get the value of apiSecret  
def mySecret = appSettings.apiSecret
```

Note: All values in `appSettings` are stored as strings. Any desired type conversion will need to be performed manually.

Any SmartApp that requires the use of API keys or other information that is sensitive in nature should use `appSettings` to store this information.

108.11 Examples

The Github page [page-params-by-href.groovy](#) shows how to pass parameters to dynamic pages using the `href` element.

Almost every SmartApp makes use of preferences to some degree. You can browse them in the IDE under the “Browse SmartApp Templates” menu.

Storing Data With State

SmartApps and Device Handlers execute in response to various Events or schedules; they are not continuously running. Since each execution is executed independently, it has no information regarding previous executions. This is often adequate for most SmartApps or Device Handlers, but sometimes, they need to remember information across executions.

For this reason, SmartApps and Device Handlers can store small amounts of data using a map-like API, and retrieve this data in later executions.

SmartApps can persist and retrieve data one of two ways - using the built-in `state` or `atomicState` objects (the details and difference between these implementations are discussed in detail in this document). Device Handlers can use the built-in `state` object, just as SmartApps can, but do not have `atomicState` available.

Warning: As discussed in the documentation below, `state` and `atomicState` should **never** be used in the same SmartApp.

Doing so can cause inconsistencies or even loss of data. At some point, this may be enforced through a compile-time check to prevent making this mistake.

109.1 Quick example

Consider this simple example that keeps track of how many time a switch is turned on:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}

def installed() {
    initialize()
}

def updated() {
    unsubscribe()
    initialize()
}

def initialize() {
    // initialize counter
}
```

```
state.switchCounter = 0

subscribe({switch}, "switch.on", incrementCounter)
}

def incrementCounter() {
    state.switchCounter = state.switchCounter + 1
    log.debug "switch has been turned on $state.switchCounter times"
}
```

As you can see, using State is straightforward - we can add and retrieve data just as we would with a map (actually, State is an implementation of `java.util.Map`, as discussed more below).

While working with State appears straightforward (and for simple use cases, it is), an understanding of the workings of State is necessary to avoid debugging headaches, inconsistent data results, and even potential data loss in certain scenarios. With this information, you can make the best use of State (and Atomic State) and save yourself and your customers a good deal of trouble that *could* be encountered.

109.2 State and Atomic State overview

There are two objects injected into every SmartApp to persist and retrieve data across executions: `state` and `atomicState` (Device Handlers only have `state` available, but an understanding of how `state` works is still important for Device Handler developers).

Here are the key features and differences between State and Atomic State. The details of both are discussed in this document, along with guidelines for understanding which to use in different situations.

State:

- State is an implementation of `java.util.Map`, making it simpler and more feature-rich to work with.
- Modifications (addition, removal, updating) to State within an execution are only persisted to external storage *after execution completes*. This makes State the more performant choice.

Atomic State:

- Atomic State is not an implementation of `java.util.Map`, so working with it is not as feature-rich as State.
- Modifications (additional, removal, updating) to Atomic State within an execution are persisted to external storage *more or less immediately*. This incurs a performance penalty when compared to State.

109.3 Persistence model

Both State and Atomic State use a database table to store values. The same table is used by both State and Atomic State.

The values are stored as JSON strings. Given the following code:

```
def initialize() {
    state.someString = "some string"
    state.someNum = 42
    state.collection = [k1: 1, k2: [n1: "nested"]]
}
```


The data stored in the database table would look like this:

Installed SmartApp ID	Name	Value
<installed-smartapp-id>	someString	“some string”
<installed-smartapp-id>	someNum	42
<installed-smartapp-id>	collection	{“k1”:1,“k2”:{“n1”:”nested”}}

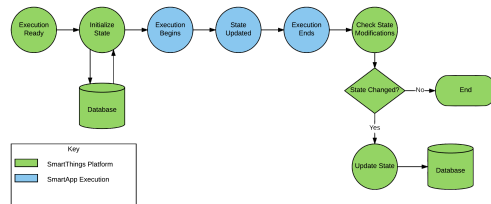
109.4 How State works

All SmartApps and Device Handlers have available to them a `state` object (it is a map) to persist data between executions.

The general flow for SmartApp state is as follows:

1. When a SmartApp or Device Handler is scheduled for execution, the `state` object is populated with the values from the database. The SmartThings platform also makes a copy of the contents of `state` prior to execution, for later comparison.
2. SmartApp or Device Handler execution begins, and can add, read, or modify the contents in the `state` object just as with any other map.
3. Execution ends. The SmartThings platform compares the `state` object at execution ends with the contents of `state` before execution began. If there are any changes (additions, removals, updates), those entries are written to the database.

This is summarized in the following diagram:



109.5 State and potential race conditions

Since `state` is initialized from persistent storage when a SmartApp executes, and is written to storage only when the application is done executing, there is the possibility that another execution *could* happen within that time window, and cause the values stored in `state` to appear inconsistent.

Consider the scenario of a SmartApp that keeps a counter of executions. Each time the SmartApp executes, it increments the counter by 1. Assume that the initial value of `state.counter` is 0.

1. An execution (“Execution 1”) occurs, and increments `state.counter` by one:

```
state.counter = state.counter + 1 // counter == 1
```

2. Another execution (“Execution 2”) occurs *before* “Execution 1” has finished. It reads `state.counter` and increments it by one:

```
state.counter = state.counter + 1 // counter == 1!!!
```

Because “Execution 1” hasn’t finished executing by the time that “Execution 2” begins, the value of `counter` is still 0!

Additionally, because the contents of `state` are only persisted when execution is complete, it’s also possible to inadvertently overwrite values (last finished execution “wins”).

To avoid this type of scenario, SmartApps can use Atomic State, which is discussed next. Atomic State writes to the data store when a value is *set*, and reads from the data store when a value is *read* - not just when the application execution initializes and completes.

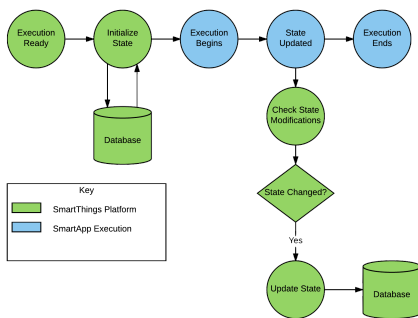
Before using Atomic State, you should read about *how to choose between State and Atomic State* (page 318).

109.6 How Atomic State works

SmartApps have available to them, in addition to `state`, also the object `atomicState`, which operates like `state` with two notable differences:

1. Atomic State does not implement `java.util.Map`.
2. When items are added or modified to Atomic State, those values are persisted more or less immediately (unlike `State`, which only persists its data when execution finishes).

The following diagram illustrates how Atomic State is initialized and updated when a SmartApp executes:



109.7 Choosing between State and Atomic State

Given the choice between `State` and Atomic State, which should you use?

In short, prefer `State` until analysis and testing shows you otherwise. The reasons for this are:

1. `State` is easier to work with, since it supports `java.util.Map`.
2. `State` is more performant than Atomic State, since it does not read or write to external storage during SmartApp execution.

You may need to use Atomic State if code that updates a value in `State` may execute at the same time as another instance of the same SmartApp, updating the same `State` key, as discussed [here](#) (page 317).

Important: Never use both Atomic State and State in the same SmartApp. This can't be emphasized enough - doing so may result in data inconsistency, data corruption, or even data loss.

109.8 What can be stored in State and Atomic State

state and atomicState values are stored as a JSON string by SmartThings.

109.8.1 Supported types

The following types are supported for storage in State and Atomic State:

- String
- long
- int
- BigDecimal
- true
- false
- null
- ArrayList
- Map

Here is an example illustrating this:

```
def initialize() {
    state.string = "string"
    state.int = 42
    state.long = now()
    state.decimal = 4.2
    state.yes = true
    state.no = false
    state.empty = null
    state.list = [1, 2, 3, 4]
    state.map = [a: 1, b: 2, c: "three"]
    runIn(60, check)
}

def check() {
    def isString = state.string instanceof String // -> true
    def isInt = state.int instanceof Integer // -> true
    def isLong = state.long instanceof Long // -> true
    def isDecimal = state.decimal instanceof BigDecimal // -> true
    def isBoolean = state.yes instanceof Boolean // -> true
    def isAlsoBoolean = state.no instanceof Boolean // -> true
    def isNull = state.empty == null // -> true
    def isList = state.list instanceof List // -> true
    def isMap = state.map instanceof Map // -> true
}
```

```
// items in map
def isMapInt = state.map.b instanceof Integer // -> true
def isMapString = state.map.c instanceof String // -> true
```

109.8.2 Other object types

SmartThings objects (like *Event* (page 1017), *Device* (page 1003), etc.) cannot be stored in State or Atomic State. If you attempt to store these objects, it will silently fail without any messages in Live Logging.

If you need to store such information on State, get the specific data you need from the object and assign it to state, like so:

```
def someEventHandler(evt) {
    state.someEvent = [name: evt.name, value: evt.value, id: evt.id]
}
```

Dates also require some care when storing in state. If you were to store a date directly, you would end up with a string representation of the date when retrieving it.

```
def initialize() {
    state.date = new Date()
    runIn(30, check)
}

def check() {
    def isDate = state.date instanceof Date // -> false
    def isString = state.date instanceof Date // -> true
}
```

If you need to store time information, consider using an epoch time stamp, conveniently available via the *now()* (page 926) method:

```
def installed() {
    state.installedAt = now()
}

def someEventHandler(evt) {
    def millisSinceInstalled = now() - state.installedAt
    log.debug "this app was installed ${millisSinceInstalled / 1000} seconds ago"

    // you can also create a Date object back from epoch time:
    log.debug "this app was installed at ${new Date(state.installedAt)}"
}
```

109.9 Working with the state object

state is an implementation of `java.util.Map`. This means you can interact with the `state` object in a SmartApp or Device Handler just as you would with any other map.

Just remember that all modifications done to `state` within a SmartApp or Device Handler are only written to external storage after the execution completes.

Important: Be sure to read the *Overview* (page 316) and *How State works* (page 317) documentation before using `state`.

109.9.1 Adding values

Add values to `state` just as you would with a map:

```
state.someKey = "some val"
state['otherKey'] = 32
```

109.9.2 Retrieving values

Get values from `state` just as you would with a map, using either dot notation or index notation (we prefer dot notation for simplicity):

```
state.someKey = "some val"
log.debug "value of state.someKey: $state.someKey"

state.someOtherKey = 42
log.debug "value of state['someOtherKey']: ${state['someOtherKey']}"
```

109.9.3 Updating values

To update the value for an existing key in `state`, simply assign a new value to it:

```
state.someKey = "some val"
log.debug "state.someKey: $state.someKey" // -> some val
state.someKey = "updated"
log.debug "state.someKey: $state.someKey" // -> updated
```

109.9.4 Removing values

Because `state` is a map, we can use the `remove()` method to remove the item:

```
state.someKey = "some val"
log.debug "state: $state" // -> [someKey: "some val"]
state.remove('someKey')
log.debug "state: $state" // -> [:]
```

109.9.5 Iterating over state

We can iterate over the values in `state` just as we would with a map, using `each()`:

```
state.keyOne = "val one"
state.keyTwo = "val two"

state.each {key, val ->
    log.debug "state key: $key, value: $val"
}
```

We can also find entries using any of Groovy's collections methods like `find()`, `findAll()`, `collect()`, etc:

```
state.key_one = "val one"
state.key_two = "val two"
state.someOther = 42

def found = state.findAll {k, v ->
    v.startsWith('key_')}

log.debug "found: $found" // -> [key_one: "val one", key_two: "val two"]
```

109.9.6 Working with collections

Working with collections in `state` is straightforward:

```
state.collection = [k1: "one", k2: "two", k3: [n1: 2, n2: 3]]
state.collection.k1 = "UPDATED"
state.k3.n1 = "ALSO UPDATED"

// [k1: "UPDATED", k2: "two", k3: [n1: 2, n2: "ALSO UPDATED"]]
log.debug "state: $state"
```

109.10 Working with the `atomicState` object

For simple use cases, working with Atomic State is just like working with State - you can assign and retrieve values just as with State. The key difference is that Atomic State does *not* implement `java.util.Map`, so using map operations like `remove()`, `forEach()`, `find()`, etc., will not work with Atomic State.

Important: Be sure to read the *Overview* (page 316), *How Atomic State works* (page 318), and *Choosing between State and Atomic State* (page 318) documentation before using `atomicState`.

109.10.1 Adding values

We can add values to Atomic State just as we do with State:

```
atomicState.someKey = "some val"
log.debug "value of atomicState.someKey: $atomicState.someKey"

atomicState.someOtherKey = 42
log.debug "value of atomicState['someOtherKey']: ${atomicState['someOtherKey']}"
```

109.10.2 Updating values

To update the value for an existing key in Atomic State, simply assign a new value to it.

Note: Updating collections in `atomicState` is a special case, and is discussed [here](#) (page 323).

```
atomicState.someKey = "some val"
log.debug "atomicState.someKey: $atomicState.someKey" // -> some val
atomicState.someKey = "updated"
log.debug "atomicState.someKey: $atomicState.someKey" // -> updated
```

109.10.3 Removing values

Removing items from Atomic State is not possible, since it does not implement `java.util.Map`. Instead, you can set the value to `null`:

```
atomicState.someExistingKey = null
```

Note that this does not remove the key from Atomic State; it simply sets the value to `null`.

109.10.4 Iterating over all values

Iterating over all items in Atomic State is not possible, because it does not implement `java.util.Map`.

109.10.5 Working with collections

Updating collections stored in Atomic State is different than working with collections in State.

Instead, you will need to assign the collection to a local variable, make changes as needed, then assign it back to `atomicState`. Here's an example:

```
def initialize() {
    atomicState.myMap = [key1: "val1"]
    log.debug "atomicState: $atomicState"

    // assign collection to local variable and update
    def temp = atomicState.myMap
    // update existing entry
    temp.key1 = "UPDATED"
    // add new entry
    temp.key2 = "val2"

    // assign collection back to atomicState
    atomicState.myMap = temp
    log.debug "atomicState: $atomicState"
}
```

109.11 Storage size limits

The contents of State and Atomic State are limited to 100,000 characters when serialized to JSON.

This should be more than sufficient for typical use cases. If you find yourself running into this limitation, you should evaluate your use case - remember, State and Atomic State are intended to persist small amounts of data across executions. It is not intended to be an unbounded or large database.

To get the character size of `state` or `atomicState`, you can do:

```
def stateCharSize = state.toString().length()
```

When the character limit has been exceeded, a `physicalgraph.exception.StateCharacterLimitExceededException` will be thrown.

Important: Remember that when using `state`, the contents are written to the external data store when the app is finished executing - not immediately on write/read from the object.

This means that if the character limit is exceeded for `state`, you won't be able to handle a `StateCharacterLimitExceededException` in your code - it will only be visible in the logs.

If using `atomicState`, which reads and writes to the external data store when the object is updated or accessed, you will be able to handle a `StateCharacterLimitExceededException` in your code.

Additional helper methods to get the remaining available size and the character limit will be added in a future release.

109.12 State in parent-child relationships

If you are attempting to access the State or Atomic State of a parent or child relationship, you may encounter a `NullPointerException`. As a workaround, you can create a method to get State or Atomic State values like this:

```
def getStateValue(key) {  
    return state[key]  
}
```

You could create a similar method to update State or Atomic State across parent-child relationships, but be careful. Because there could be multiple children for a parent SmartApp, for example, updating the parent's State or Atomic State from the children may introduce additional complexity and opportunity for race conditions and inconsistent values.

109.13 Summary

- State and Atomic State allow developers to persist data across executions.
- State and Atomic State are both available to SmartApps; only State is available to Device Handlers.
- State and Atomic State use the same underlying database table.
- State values are persisted after the current execution ends. Atomic State values are persisted immediately.
- State implements `java.util.Map`, Atomic State does not.
- State and Atomic State allow for the storage of strings, numbers, booleans, null values, lists, and maps.
- Never mix State and Atomic State in the same SmartApp.
- Prefer State unless analysis and testing shows Atomic State is necessary.
- State and Atomic State are limited to 100,000 characters of data (when serialized to JSON) per installed SmartApp or Device Handler.

Events and Subscriptions

Turn on a light when a door opens. Turn the lights off at sunrise. Send a message if a door opens when you're not home. These are all examples of event-handler SmartApps. They follow a common pattern - subscribe to some Event, and take action when the Event happens.

This section will discuss Events and how you can subscribe to them in your SmartApp.

110.1 Subscribe to specific device Events

The most common use case for Event subscriptions is for device Events:

```
1 preferences {
2   section {
3     input "theSwitch", "capability.switch"
4   }
5 }
6
7 def install() {
8   subscribe(theSwitch, "switch.on", switchOnHandler)
9 }
10
11 def switchOnHandler(evt) {
12   log.debug "switch turned on!"
13 }
```

The handler method must accept an Event parameter.

Refer to the [Event](#) (page 1017) API documentation for more information about the Event object.

You can find the possible Events to subscribe to by referring to the Attributes column for a capability in the [Capabilities Reference](#) (page 655). The general form we use is “<attributeName>.<attributeValue>”. If the attribute does not have any possible values (for example, “battery”), you would just use the attribute name.

In the example above, the switch capability has the attribute “switch”, with possible values “on” and “off”. Putting these together, we use “switch.on”.

110.2 Subscribe to all device Events

You can also subscribe to all states by just specifying the attribute name:

```
subscribe(theSwitch, "switch", switchHandler)

def switchHandler(evt) {
    if (evt.value == "on") {
        log.debug "switch turned on!"
    } else if (evt.value == "off") {
        log.debug "switch turned off!"
    }
}
```

In this case, the `switchHandler` method will be called for both the “on” and “off” Events.

110.3 Subscribe to multiple devices

If your SmartApp allows multiple devices, you can subscribe to Events for all the devices:

```
preferences {
    section {
        input "switches", "capability.switch", multiple: true
    }
}

def installed() {
    subscribe(switches, "switch", switchesHandler)
}

def switchesHandler(evt) {
    log.debug "one of the configured switches changed states"
}
```

110.4 Subscribe to Location Events

In addition to subscribing to device Events, you can also subscribe to Events for the user’s Location.

You can subscribe to the following Location Events:

mode Triggered when the mode changes.

position Triggered when the geofence position changes for this Location. Does not get triggered when the fence is widened or narrowed - only fired when the position changes.

sunset Triggered at sunset for this Location.

sunrise Triggered at sunrise for this Location.

sunriseTime Triggered around sunrise time. Used to get the time of the next sunrise for this Location.

sunsetTime Triggered around sunset time. Used to get the time of the next sunset for this Location.

Pass in the Location property automatically injected into every SmartApp as the first parameter to the subscribe method.

```
subscribe(location, "mode", modeChangeHandler)

// shortcut for mode change handler
subscribe(location, modeChangeHandler)

subscribe(location, "position", positionChange)
subscribe(location, "sunset", sunsetHandler)
subscribe(location, "sunrise", sunriseHandler)
subscribe(location, "sunsetTime", sunsetTimeHandler)
subscribe(location, "sunriseTime", sunriseTimeHandler)
```

Refer to the [Sunset and Sunrise](#) section for more information about sunrise and sunset.

110.5 The Event object

Event-handler methods must accept a single parameter, the Event itself.

Refer to the [Event](#) (page 1017) API documentation for more information.

A few of the common ways of using the Event:

```
def eventHandler(evt) {
    // get the event name, e.g., "switch"
    log.debug "This event name is ${evt.name}"

    // get the value of this event, e.g., "on" or "off"
    log.debug "The value of this event is ${evt.value}"

    // get the Date this event happened at
    log.debug "This event happened at ${evt.date}"

    // did the value of this event change from its previous state?
    log.debug "The value of this event is different from its previous value: ${evt.isStateChange()}"
}
```

Note: The contents of each Event instance will vary depending on the exact Event. If you refer to the Event reference documentation, you will see different value methods, like “floatValue” or “dateValue”. These may or may not be populated depending on the specific Event, and may even throw exceptions if not applicable.

110.6 See also

- [Sunset and Sunrise](#)
- [Event](#) (page 1017) API Documentation
- [Location](#) (page 1036) API Documentation
- [Interacting with Devices](#)

Working with Devices

SmartApps almost always interact with devices. We often need to get information about a specific device (is this switch on?), or send a device a command (turn this switch off).

111.1 Device overview

Devices are the “things” that SmartApps interact with. Devices may support one or many capabilities.

Capabilities represent the things a device knows (attributes) and the things they can do (commands). They are an abstraction that allows us to work with many different manufacturer’s devices transparently.

To build a flexible SmartApp, we should write our SmartApp to work with any device that supports a given capability. We don’t want to write a SmartApp that only works with a specific manufacturer’s switch, for example. We want to write an app that works with any device that supports the switch capability.

111.2 Preferences—selecting the devices

To allow the user to select devices that support a given capability, we use the preferences input element:

```
preferences {
  section {
    input "presenceSensors", "capability.presenceSensor"
  }
}
```

The above example will allow the user to select any device that supports the presence sensor capability. This could be a mobile phone, or a [SmartSense presence sensor](#). We don’t care about the specific device - we just declare we want a device that supports the presence sensor capability.

You can refer to the [Capabilities Reference](#) (page 655) for information on all the supported capabilities. The “Preferences Reference” column tells you what to use in your preferences for a given capability.

111.3 Interacting with devices

After you have declared the devices your SmartApp needs to interact with, a Device object instance will be available in your SmartApp, with the name that you provided.

```
preferences {
    section {
        input "theSwitch", "capability.switch"
    }
}

def someEventHandler(evt) {
    theSwitch.on()
}
```

111.4 Device attributes

Attributes represent the state of a device. A device that supports the “temperatureMeasurement” capability has a “temperature” attribute, for example.

Attributes have state - the “temperature” attribute has an associated *State* (page 1040) object that contains information about the temperature (its value, the date it was recorded, etc.).

Attribute data is stored in the SmartThings Cloud and updated when the device reports its status.

111.5 Device commands

Devices may expose one or many commands. Commands are the things that devices can do. A switch supports the “on” and “off” commands, that turn the switch “on” and “off”, respectively.

Not all devices have commands. Commands typically perform some sort of physical actuation (turn a switch on, or unlock a lock, for example). A humidity sensor has nothing to physically actuate, for example.

111.6 Getting device current values

Information about the most recently reported device attribute state can be retrieved in two ways:

currentState() (page 1006) and *<attribute name>State* (page 1004) return a *State* (page 1040) object that encapsulates the most recently reported state of the device.

```
preferences {
    section() {
        input "tempSensor", "capability.temperatureMeasurement"
    }
}

def someEventHandler(evt) {
```

```

def currentState = tempSensor.currentState("temperature")
log.debug "temperature value as a string: ${currentState.value}"
log.debug "time this temperature record was created: ${currentState.date}"

// shortcut notation - temperature measurement capability supports
// a "temperature" attribute. We then append "State" to it.
def anotherCurrentState = tempSensor.temperatureState
log.debug "temperature value as an integer: ${anotherCurrentState.integerValue}"
}

```

`latestValue()` (page 1015), `currentValue()` (page 1006), and `current<Uppercase attribute name>` (page 1005) returns the most recently reported attribute value. These can be used interchangeably; they all do the same thing.

```

preferences {
    section() {
        input "myLock", "capability.lock"
    }
}

def someEventHandler(evt) {
    def currentValue = myLock.currentValue("lock")
    log.debug "the current value of myLock is $currentValue"

    def latestValue = myLock.latestValue("lock")
    log.debug "the latest value of myLock is $latestValue"

    // Lock capability has "lock" attribute.
    // <deviceName>.current<uppercase attribute name>:
    def anotherCurrentValue = myLock.currentLock
    log.debug "the current value of myLock using shortcut is: $anotherCurrentValue"
}

```

Important: The current or latest state for an attribute value is the *most recent value the device has reported to SmartThings*. It is not calculated by polling or otherwise directly communicating with the device.

For example, `someDevice.currentValue('someAttribute')` will get the most recently reported value for the specified attribute. If the device has malfunctioned, or the SmartThings Hub has gone offline, it is possible that the value returned is not consistent with the physical status of the device.

111.7 Querying event history

To get a list of Events in reverse chronological order (newest first), use the `events()` method:

```

// returns the last 10 by default
myDevice.events()

// use the max option to get more results
myDevice.events(max: 30)

```

To get a list of Events in reverse chronological order (newest first) since a given date, use the `eventsSince` method:

```
// get all events for this device since yesterday (maximum of 1000 events)
myDevice.eventsSince(new Date() - 1)

// get the most recent 20 events since yesterday
myDevice.eventsSince(new Date() - 1, [max: 20])
```

To get a list of Events between two dates, use the `eventsBetween` method:

```
// get all events between two days ago and yesterday (up to 1000 events)
// returned events sorted in inverse chronological order (newest first)
myDevice.eventsBetween(new Date() - 2, new Date() - 1)

// get the most recent 50 events in the last week
myDevice.eventsBetween(new Date() - 7, new Date(), [max: 50])
```

Similar date-constrained methods exist for getting State information for a device.

Refer to the full *Device* (page 1003) API documentation for more information.

111.8 Sending commands

SmartApps often need to send commands to a device - tell a switch to turn on, or a lock to unlock, for example.

The commands available to your device will vary by device. You can refer to the *Capabilities Reference* (page 655) to see the available commands for a given capability.

Sending a command is as simple as calling the command method on the device:

```
myLock.lock()
myLock.unlock()
```

Some commands may expect parameters. All commands can take an optional map parameter, as the last argument, to specify delay time in milliseconds to wait before the command is sent to the device:

```
// wait two seconds before sending on command
mySwitch.on([delay: 2000])
```

Note: Because specific devices *can* provide more commands than its supported capabilities, it is possible to have more available commands than the capability declares. As a best practice, you should write your SmartApp to the capabilities specification, and not to any specific device. If, however, you are writing a SmartApp for a very specific case, and are willing to forgo the flexibility, you may make use of this ability.

111.9 Interacting with multiple devices

If you specified `multiple:true` in your device preferences, the user may have selected more than one device. Your device instance will refer to a list of objects if this is the case.

You can send commands to all the devices without needing to iterate over each one:


```
preferences {
    section {
        input "switches", "capability.switch", multiple: true
    }
}

def someEventHandler(evt) {
    log.debug "will send the on() command to ${switches.size()} switches"
    switches.on()
}
```

You can also retrieve state and event history for multiple devices, using the methods discussed above. Instead of single values or objects, they will return a list of values or objects.

Here's a simple example of getting all switch state values and logging the switches that are on:

```
preferences {
    section {
        input "switches", "capability.switch", multiple: true
    }
}

def someEventHandler(evt) {
    // returns a list of the values for all switches
    def currSwitches = switches.currentSwitch

    def onSwitches = currSwitches.findAll { switchVal ->
        switchVal == "on" ? true : false
    }

    log.debug "${onSwitches.size()} out of ${switches.size()} switches are on"
}
```

111.10 See also

- *Capabilities Reference* (page 655)
- *Preferences and Settings* (page 285)
- *Events and Subscriptions* (page 325)
- *Device* (page 1003) API Documentation
- *Event* (page 1017) API Documentation
- *State* (page 1040) API Documentation

Modes

SmartThings allows users to specify that SmartApps only execute when in certain *modes*.

112.1 Overview

Modes can be thought of as behavior filters for the smart home. Users can change how things act or behave based on the mode you're in. For example:

- When in “Home” mode, motion should turn on a light.
- When in “Away” mode, motion should send a text message and turn on an alarm.

SmartThings comes with a few pre-configured modes, such as “Home”, “Away”, and “Night”. Users can also create their own modes for each Location.

112.2 Getting the current Mode

You can get the current mode by using the `mode` or `currentMode` property on the `location` in a SmartApp:

```
def currMode = location.mode // "Home", "Away", etc.
log.debug "current mode is $currMode"

def anotherWay = location.currentMode
log.debug "current mode is $anotherWay"
```

112.3 Getting all Modes

You can get a list of all the modes for the Location the SmartApp is installed into:

```
def allModes = location.modes // ex: [Home, Away, Night]
log.debug "all modes for this location: $allModes"
```

112.4 Setting the Mode

You can use `setLocationMode()` or `location.setMode()` to set the mode for the Location:

```
setLocationMode("Away")
```

```
location.setMode("Away")
```

These methods will raise an error if the mode specified does not exist for the Location.

112.5 Allowing users to select Modes

In the SmartApp preferences block, you can specify that the user select a mode by using the "mode" input type:

```
input "modes", "mode", title: "select a mode(s)", multiple: true
```

This will allow the user to select a mode (or multiple modes), and the SmartApp can then vary its behavior based upon the mode(s) selected.

You can also use the `mode()` method to allow a user to select a mode that this SmartApp will execute for:

```
mode(title: "Set for specific mode(s)")
```

The SmartApp will then only execute when in the selected mode, without any action needed by the developer to determine the correct mode.

You can learn more about the various ways to allow a user to select a mode [here](#) (page 300).

112.6 Mode events

You can listen for a mode change by subscribing to the "mode" on the `location` object:

```
def installed() {
    subscribe(location, "mode", modeChangeHandler)
}

def modeChangeHandler(evt) {
    log.debug "mode changed to ${evt.value}"
}
```

In the example above `modeChangeHandler()` will be called whenever the mode changes for the Location this SmartApp is installed into.

112.7 Example

The following example is a simplified version of the “Scheduled Mode Change” SmartApp. You can view the SmartApp in the IDE templates for the full example.

This example shows how to use the "mode" input type to ask the user to select a mode, and then (based on the user-defined schedule), changes the mode as specified.

```

preferences {
    section("At this time every day") {
        input "time", "time", title: "Time of Day"
    }
    section("Change to this mode") {
        input "newMode", "mode", title: "Mode?"
    }
}

def installed() {
    initialize()
}

def updated() {
    unschedule()
    initialize()
}

def initialize() {
    schedule(time, changeMode)
}

def changeMode() {
    log.debug "changeMode, location.mode = $location.mode, newMode = $newMode, location.modes = $location.modes"

    if (location.mode != newMode) {
        if (location.modes?.find{it.name == newMode}) {
            setLocationMode(newMode)
        } else {
            log.warn "Tried to change to undefined mode '${newMode}'"
        }
    }
}

```

In the `changeMode()` method above, there are a few things worth calling out.

First, notice we first check if we are already in the mode specified - if we are, we don't do anything:

```
if (location.mode != newMode)
```

If we do need to change the mode, we first verify that the mode actually exists. This ensures that we don't try and set the mode to one that does not exist for the Location.

```
if (location.modes?.find{it.name == newMode})
```

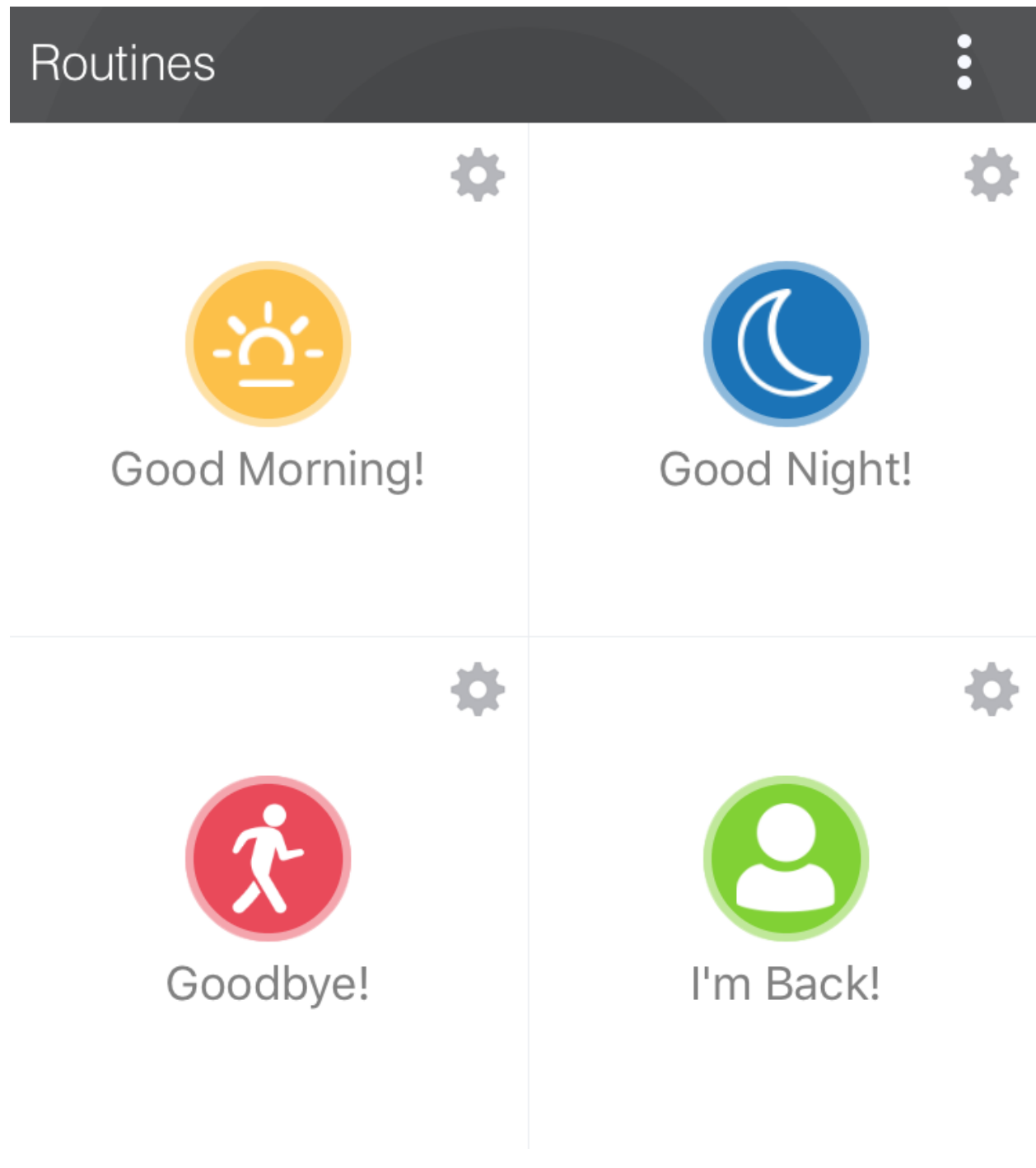
112.8 Further reading

- *Mode Input* (page 300)

- *Location Object* (page 1036)
- *Mode Object* (page 1039)

Routines

Routines (or *Hello Home Actions* in older mobile apps) allow certain things to happen when the Routine is invoked.



113.1 Overview

Routines allow for certain things to happen whenever it executes. SmartThings comes with a few Routines already installed:

- Good Morning! - You or the house is waking up

- Good Night! - You or the house is going to sleep
- Goodbye! - You're leaving the house
- I'm Back! - You've returned to the house

Each Routine can be configured to do certain things. For example, when “I'm Back!” executes, you can set the Mode to “Home”, unlock doors, adjust the thermostat, etc.

Routines exist for each Location in a SmartThings account.

113.2 Get available Routines

You can get the Routines for the Location the SmartApp is installed into by accessing the `helloHome` object on the `location`:

```
def actions = location.helloHome?.getPhrases()*.label
```

Tip: If the above code example, with the `?` and `*` operator looks foreign to you, read on.

The `?` operator allows us to safely avoid a `NullPointerException` should `helloHome` be null. It's one of Groovy's niceties that allows us to avoid wrapping calls in `if(something != null)` blocks. Read more about it [here](#).

The `*` operator is called the *spread operator*, and it invokes the specified action (get the label, in the example above) on all items in a collection, and collects the result into a list. Read more about it [here](#).

113.3 Execute Routines

To execute a Routine, you can call the `execute()` method on `helloHome`:

```
location.helloHome?.execute("Good Night!")
```

113.4 Allowing users to select Routines

A SmartApp may want to allow a user to execute certain Routines in a SmartApp. Since the Routines for each Location will vary, we need to get the available Routines, and use them as options for an `enum` input type.

This needs to be done in a dynamic preferences page, since we need to execute some code to populate the available actions:

```
preferences {
    page(name: "selectActions")
}

def selectActions() {
    dynamicPage(name: "selectActions", title: "Select Hello Home Action to Execute", install: true,
```

```
// get the available actions
def actions = location.helloHome?.getPhrases()*label
if (actions) {
    // sort them alphabetically
    actions.sort()
    section("Hello Home Actions") {
        log.trace actions
        // use the actions as the options for an enum input
        input "action", "enum", title: "Select an action to execute", options: actions
    }
}
}
```

You can read more about the enum input type and dynamic pages [here](#) (page 285).

You can then access the selected phrase like so:

```
def selectedAction = settings.action
```

113.5 Routine Events

When a Routine is executed, a "routineExecuted" event is created for that Location. Here's how you can subscribe to a Routine being executed in a SmartApp:

```
def initialize() {
    // subscribe to the "routineExecuted" event on the location
    subscribe(location, "routineExecuted", routineChanged)
}

def routineChanged(evt) {
    log.debug "routineChanged: $evt"

    // name will be "routineExecuted"
    log.debug "evt name: ${evt.name}"

    // value will be the ID of the SmartApp that created this event
    log.debug "evt value: ${evt.value}"

    // displayName will be the name of the routine
    // e.g., "I'm Back!" or "Goodbye!"
    log.debug "evt displayName: ${evt.displayName}"

    // descriptionText will be the name of the routine, followed by the action
    // e.g., "I'm Back! was executed" or "Goodbye! was executed"
    log.debug "evt descriptionText: ${evt.descriptionText}"
}
```

113.6 Example

This example simply shows executing a selected Routine when a switch turns on, and another action when a switch turns off:

```
preferences {
    page(name: "configure")
}

def configure() {
    dynamicPage(name: "configure", title: "Configure Switch and Phrase", install: true, uninstall:
        section("Select your switch") {
            input "theswitch", "capability.switch", required: true
        }

    def actions = location.helloHome?.getPhrases()*.label
    if (actions) {
        actions.sort()
        section("Hello Home Actions") {
            log.trace actions
            input "onAction", "enum", title: "Action to execute when turned on", options: actions
            input "offAction", "enum", title: "Action to execute when turned off", options: actions
        }
    }
}

def installed() {
    log.debug "Installed with settings: ${settings}"
    initialize()
}

def updated() {
    log.debug "Updated with settings: ${settings}"
    unsubscribe()
    initialize()
}

def initialize() {
    subscribe(theswitch, "switch", handler)
    subscribe(location, "routineExecuted", routineChanged)
    log.debug "selected on action $onAction"
    log.debug "selected off action $offAction"
}

def handler(evt) {
    if (evt.value == "on") {
        log.debug "switch turned on, will execute action ${settings.onAction}"
        location.helloHome?.execute(settings.onAction)
    } else {
        log.debug "switch turned off, will execute action ${settings.offAction}"
        location.helloHome?.execute(settings.offAction)
    }
}

def routineChanged(evt) {
    log.debug "routineChanged: $evt"
    log.debug "evt name: ${evt.name}"
    log.debug "evt value: ${evt.value}"
    log.debug "evt displayName: ${evt.displayName}"
    log.debug "evt descriptionText: ${evt.descriptionText}"
}
```

113.7 Further reading

- *Preferences and Settings Guide* (page 285)

Scheduling

SmartApps and Device Handlers often need to schedule certain actions to take place at a given point in time. For example, an app may want to turn off the lights five minutes after someone leaves. Or, an app may want to turn on the lights every day at a certain time.

114.1 Overview

Broadly speaking, there are a few different ways we might want to schedule something to happen:

- Do something after a certain duration of time from now.
- Do something once at a certain time in the future.
- Do something on a recurring schedule.

We'll look at each scenario in detail, and at the methods SmartThings makes available to address these requirements.

Note: When using the scheduler APIs, the schedule will be created using the time zone of the SmartApp's Location.

114.2 Schedule from now—`runIn()`

A SmartApp may want to take some action within a certain duration of time after some event has occurred. Consider a few examples:

- Turn a light off two minutes after a door closes.
- Adjust the thermostat ten minutes after everyone leaves.
- If a door opens and is not shut after five minutes, send a notification.

All these scenarios follow a common pattern: when a certain event happens, take some action after a given duration of time. This can be accomplished this by using the `runIn()` (page 928) method.

The `runIn()` method executes a specified handler method after a given number of seconds have elapsed.

```
def someEventHandler(evt) {
    // execute handler in five minutes from now
    runIn(60*5, handler)
}

def handler() {
    theswitch.off()
}
```

By default, if a method is scheduled to run in the future, and then if another call to `runIn()` with the same method is made, the last one overwrites the previously scheduled method. This is usually preferable.

Consider a situation where we have a switch scheduled to turn off after five minutes of a door closing:

- First, the door closes at 2:50 and we schedule the switch to turn off after five minutes (2:55).
- Then, two minutes later (2:52), the door opens and closes again - another call to `runIn()` will be made to schedule the switch to turn off in five minutes from now (2:57).

By default, there will now be *one* scheduled execution, at 2:57. And in this scenario, that is preferable.

But if don't want the most recent scheduled handler to execute, we can specify `[overwrite: false]`:

```
def someEventHandler(evt) {
    runIn(300, handler, [overwrite: false])
}

def handler() {
    // need to handle multiple calls since overwrite:false specified
}
```

We would now have two schedules to turn off the switch - one at 2:55, and one at 2:57. So, if you do specify `[overwrite: false]`, be sure to write your handler so that it can handle multiple calls.

Note: It is important to note that you should not rely on `runIn()` being called in *exactly* the specified number of seconds. SmartThings will *attempt* to execute the method within a minute of the time specified, but cannot guarantee it. See the *Best practices* (page 352) topic below for more information.

114.3 Run once in the future—`runOnce()`

Some SmartApps may need to schedule certain actions to happen *once* at a specific time and date. `runOnce()` (page 934) handles this case.

You can pass a `Date` object or a Java ISO-8601 formatted string ¹.

¹ You may notice that some of the scheduling APIs accept a string to represent the the date/time to be executed. This is a result of when you define a preference input of the "time" type, it uses a `String` representation of the value entered. When using this value later to set up a schedule, the APIs need to be able to handle this type of argument. When simply using the input from preferences, you don't need to know the details of the specific date format being used. But, if you wish to use the APIs with string inputs directly, you will need to understand their expected format. SmartThings uses the Java standard format of "yyyy-MM-dd'T'HH:mm:ss.SSSZ". More technical readers may recognize this format as ISO-8601 (Java does not fully conform to this format, but it is very similar). Full discussion of this format is beyond the scope of this documentation, but a few examples may help: "January 09, 2015 3:50:32 GMT-6 (Central Standard Time)" converts to "2015-01-09T15:50:32.000-0600", and "February 09, 2015 3:50:32:254 GMT-6 (Central Standard Time)" converts to "2015-02-09T15:50:32.254-0600" For more information about date formatting, you can review the [SimpleDateFormat JavaDoc](#).

```

preferences {
    input "executeTime", "time", title: "enter a time to execute every day"
}

def initialized() {
    // execute once at the time specified by the user
    runOnce(executeTime, handler)

    // execute tomorrow at the current time
    runOnce(new Date() + 1, handler)
}

def handler() {
    log.debug "handler executed at ${new Date()}"
}

```

Like `runIn()`, you can also specify the overwrite behavior of `runOnce()`:

```
runOnce(new Date() + 1, handlerMethod, [overwrite: false])
```

114.4 Run on a recurring schedule

Often, there is a need to schedule a job to run on a specific schedule. For example, maybe you want to turn the lights off at 11 PM every night. Or, you might need to execute a certain action every X minutes.

SmartThings provides the `schedule()` (page 935) and various `runEvery*()` methods to allow you to create recurring schedules.

The various `schedule()` methods follow a similar form - they take an argument representing the desired schedule, and the method to be called on this schedule.

Note: If a method is already scheduled, and later you call `schedule()` with that method, then that method will be executed as per the new schedule.

114.4.1 Schedule once per day

Use the `schedule()` method to execute a handler method every day at a certain time:

```

preferences {
    input "theTime", "time", title: "Time to execute every day"
}

def initialize() {
    schedule(theTime, handler)
}

// called every day at the time specified by the user
def handler() {
    log.debug "handler called at ${new Date()}"
}

```

You can also use `schedule()` with a `Date` object. Only the time portion of the `Date` will be used to derive the schedule.

```
// execute every day at the current time
schedule(new Date(), handler)
```

Finally, you can pass a `Long` representing the desired time in milliseconds (using [Unix time](#)) to `schedule()`:

```
def someEventHandler(evt) {
    // call handlerMethod every day, at two minutes from the current time
    schedule(now() + 120000, handlerMethod)
}

def handlerMethod() {
    ...
}
```

114.4.2 Schedule every X minutes or hours

For common recurring schedules, SmartThings provides a few convenience APIs that we can use.

These methods work by creating a random start time in X minutes or hours, and then every X minutes or hours after that. For example, `runEvery5Minutes(handlerMethod)` will execute `handlerMethod()` at a random time in the next five minutes, and then run every five minutes from then.

These methods have the advantage of randomizing the start time for schedules, which reduces the load on the SmartThings scheduler, and results in better performance for end users. As such, these methods should be preferred over cron expressions when available.

The currently available methods are:

- [runEvery1Minute\(\)](#) (page 929)
- [runEvery5Minutes\(\)](#) (page 930)
- [runEvery10Minutes\(\)](#) (page 931)
- [runEvery15Minutes\(\)](#) (page 931)
- [runEvery30Minutes\(\)](#) (page 932)
- [runEvery1Hour\(\)](#) (page 933)
- [runEvery3Hours\(\)](#) (page 933)

Using these methods is similar to other scheduling methods:

```
def initialize() {
    runEvery5Minutes(handlerMethod)
}

def handlerMethod() {
    log.debug "handlerMethod called at ${new Date()}"
}
```


114.4.3 Schedule using cron

Important: Prefer the `runEvery*()` methods to creating your own cron schedule when possible. These methods are documented above in the *Schedule every X minutes or hours* (page 348) section.

Scheduling jobs to execute at a particular time is useful, but what if, for example, we want a method to execute at fifteen minutes past the hour, every hour? SmartThings allows you to pass a cron expression to the `schedule()` method to accomplish this.

```
def initialize() {
    // execute handlerMethod every hour on the half hour.
    schedule("0 30 * * * ?", handlerMethod)
}

def handlerMethod() {
    ...
}
```

A cron expression is a way to specify a recurring schedule, based on the UNIX cron tool. The cron expression supported by SmartThings is a string of six or seven fields, separated by white space. The *seconds* field is the left most field. The below table describes these fields.

Field	Allowed Values	Required	Allowed Wildcards
Seconds	0-59	Yes	*
Minutes	0-59	Yes	, - * /
Hours	0-23	Yes	, - * /
Day of Month	1-31	Yes	, - * ? / L W
Month	1-12 or JAN-DEC	Yes	, - * /
Day of Week	1-7 or SUN-SAT	Yes	, - * ? / L
Year	empty, 1970-2099	No	, - * /

Allowed wildcards are:

- , (comma) is used to specify additional values. For example, SAT,SUN,MON in the Day of Week field means “the days Saturday, Sunday, and Monday.”
- – (hyphen) is used to specify ranges. For example, 5–7 in the Hours field means “the hours 5, 6 and 7”.
- * (asterisk) is used to specify all values in the field. For example, * in the Hours field means *every* hour.
- ? (question mark) is used to specify any value. For example, ? in the Day of Week field means *regardless* of what the day of the week is.
- / (forward slash) is used to specify increments. For example, 5/15 in the Minutes field means “the minutes 5, 20, 35, and 50”.
- L is used to specify the last day of the month when used in the Day of Month field and the last day of the week when used in the Day of Week fields.
- W is used to specify a weekday (Monday-Friday) that is nearest to the given day when used in the Day of Month field. For example, if you specify 21W in the Day of Month field, it means: “the nearest weekday to the 21st of the month”. So if the 21st is a Saturday, the trigger will fire on Friday the 20th. If the 21st is a Sunday, the trigger will fire on Monday the 22nd. If the 21st is a Tuesday, then it will fire on Tuesday the 21st. However if you specify 1W as the value for day-of-month, and the 1st is a Saturday, the trigger will fire on Monday the 3rd, and not on Friday, as it will not cross over the boundary of a month. The W character can only be specified when the day-of-month is a single day, not a range or list of days.

Warning: You cannot specify both the *Day of Month* and the *Day of Week* fields in the same cron expression. If you specify one of these fields, the other one must be ?.

Here is an example with the two fields, i.e., the *Day of Month* and the *Day of Week*. In the table below cases A and C are invalid.

Case	Day of Month	Day of Week	Cron Interpretation
A	*	MON	Every day of month <i>and</i> every Monday
B	*	?	Every day of month <i>and</i> whatever be the day of week
C	23	*	Every 23rd of month <i>and</i> every day of week
D	?	*	Whatever be the day of month <i>and</i> every day of week

We recommend that you test your cron expression before using it in a SmartApp or Device Handler. The cron expression test tool we use is <http://www.cronmaker.com/>.

Note: Cron jobs are only allowed to run at a rate of 1 minute or slower. If your cron expression runs faster than once per minute, it will be limited to a one minute interval. For more information, see this [community post](#).

High volume cron schedules are encouraged to specify a random seconds field. This helps to avoid a large number of scheduled executions being queued up at the same time. If you can, use a random second.

Here are some common examples for recurring schedules using cron:

Expression Description	Description
<code>schedule("12 30 * * * ?", handler)</code>	Execute <code>handler()</code> every hour on the half hour (using a randomly chosen seconds field of 12)
<code>schedule("23 0/7 * * * ?", handler)</code>	Execute <code>handler()</code> every 7 minutes beginning at 0 minutes after the hour (using a randomly chosen seconds field of 23)
<code>schedule("0 0/5 10-11 * * ?", handler)</code>	Execute <code>handler()</code> every 5 minutes beginning at 0 minutes after the hour, between the hours of 10 and 11 AM, at 0 seconds past the minute
<code>schedule("48 25 10 ? * MON-FRI", handler)</code>	Execute <code>handler()</code> at 10:25 AM Monday through Friday (using a randomly chosen seconds field of 48)

Warning: Note how you use * as it may unwittingly lead to high-frequency schedules. You may have intended to use ?. Note the difference between *, which means “every” and ?, which means “any”. For example, * */5 * * * ? means every 5th minute, run 60 times within that minute. That’s almost surely not what you want, and SmartThings will not execute your schedule that frequently (see below). If you were trying to execute every X minutes, it would look like this: 0 0/X * * * ? where X is the minute value.

114.5 Passing data to the handler method

Sometimes it is useful to pass data to the handler method. This is possible by passing in a map as the last argument to the various schedule methods with `data` as the key and another map as the value.

```
def someEventHandler(evt) {
    runIn(60, handler, [data: [flag: true]])
}

def handler(data) {
    if (data.flag) {
```

```

        theswitch.off()
    }
}

```

By passing data directly to the handler method, you can avoid having to store data in the SmartApp or Device Handler state. The following scheduling methods support passing data to their handler methods:

- `runIn()`
- `runOnce()`
- `schedule()`
- All `runEveryXMinutes()` methods
- All `runEveryXHours()` methods

Note: To also specify the overwrite flag, pass it as an additional property in the map: `[overwrite: false, data: [foo: 'bar']]`.

Similar to state, *only data that can be serialized to JSON* (page 319) can be passed to the handler.

The amount of data is limited to 2500 characters after being serialized. If this limit is exceeded, a `physicalgraph.exception.DataCharacterLimitExceededException` exception will be thrown, and the schedule will not be created.

114.6 Removing scheduled executions

You can remove scheduled executions using the `unschedule()` (page 946) method:

```

def initialize() {
    // schedule execution every 5 minutes
    runEvery5Minutes(handler)
}

def someEventHandler(evt) {
    // remove the scheduled execution
    unschedule(scheduledHandler)
}

def handler() {
    log.debug "in handler, current time is ${new Date()}"
}

```

This will remove schedules created with any of the scheduling methods (`runIn()`, `runOnce()`, and `schedule()`).

You can also call `unschedule()` with no arguments to remove all schedules:

```

// remove all scheduled executions for this SmartApp install
unschedule()

```

Note: Due to the way that the scheduling service is currently implemented, `unschedule()` is a fairly expensive operation, and may take many seconds to execute.

114.7 Viewing schedules in the IDE

You can view schedules for any installed SmartApp in the IDE.

Note: Schedules can only be viewed for SmartApps installed via the mobile client. Schedules for Device Handlers and SmartApps installed via the IDE simulator can not be viewed.

1. In the IDE, navigate to *Locations*.
2. Select the Location the SmartApp is installed into.
3. Click the *List SmartApps* link:

Installed SmartApps

List SmartApps

4. Click the name of the SmartApp you wish to view the schedules for.

You will then see various information about the installed SmartApp, including the scheduled executions:

Scheduled Jobs

Handler	Next Run Time	Prev Run Time	Status	Schedule
runInHandler	2016-03-16 12:03:18 PM CDT		PENDING	Once
cronHandler	2016-03-16 12:05:00 PM CDT		PENDING	0 0/5 * * * ? America/Chicago
timeHandler	2016-03-16 1:10:00 PM CDT		PENDING	0 10 13 * * ? * America/Chicago
anotherCronHandler	2016-03-17 10:00:00 AM CDT		PENDING	0 0/5 10-11 * * ? America/Chicago
weekdayCronHandler	2016-03-17 10:24:00 AM CDT		PENDING	0 24 10 ? * MON-FRI America/Chicago

You can view all the scheduled jobs, including the next scheduled run time, the status, and the schedule.

You can also view the SmartApp job history, which shows the previous executions and the scheduled vs. actual execution time, the delay between the scheduled time and actual time, and the total execution time for the handler method:

Job History

Handler	Scheduled Time	Actual Time	Delay (msec)	Execution (msec)
cronHandler	2016-03-13 11:55:00 AM CDT	2016-03-13 11:55:00.011 AM CDT	11	3471
cronHandler	2016-03-13 11:50:00 AM CDT	2016-03-13 11:50:00.014 AM CDT	14	131
cronHandler	2016-03-13 11:45:00 AM CDT	2016-03-13 11:45:00.014 AM CDT	14	21
cronHandler	2016-03-13 11:40:00 AM CDT	2016-03-13 11:40:00.012 AM CDT	12	2192

114.8 Best practices

When using any of the scheduling APIs, it's important to understand some limitations and best practices.

114.8.1 Avoid chained `runIn()` calls

Use `runIn()` to schedule one-time executions, not recurring schedules.

For example, do **not** do this:

```
def initialize() {
    runIn(60, handler)
}

def handler() {
    // do something here

    // schedule to run again in one minute - this is an antipattern!
    runIn(60, handler)
}
```

The above example uses a chained `runIn()` pattern to create a recurring schedule to execute every minute.

This pattern is prone to failure, because any single scheduled execution failure that results in `handler()` not being called means it will not be able to reschedule itself. One failure causes the whole chain to collapse.

If you need a recurring schedule, use `cron`.

Note: Using a chained `runIn()` pattern can be acceptable for certain short-running tasks, such as gradually dimming a bulb. But for anything long-running, use `cron`.

114.8.2 Prefer `runEvery*()` over `cron`

Use any of the `runEvery*()` (page 348) methods instead of creating your own `cron` schedule when possible.

114.8.3 Execution time may not be in exact seconds

SmartThings will try to execute your scheduled job at the specified time, but cannot guarantee it will execute at that exact moment. As a general rule of thumb, you should expect that your job will be called within the minute of scheduled execution. For example, if you schedule a job at 5:30:20 (20 seconds past 5:30) to execute in five minutes, we expect it to be executed at some point in the 5:35 minute.

114.8.4 Do not aggressively schedule

Every scheduled execution incurs a cost to launch the SmartApp, and counts against the *Rate Limits* (page 609). While there are some limitations in place to prevent excessive scheduling, it's important to note that excessive polling or scheduling is discouraged. It is one of the items we look for when reviewing community-developed SmartApps or device-type handlers.

114.8.5 `unschedule()` is expensive

As discussed above, `unschedule()` is currently a potentially expensive operation.

We plan to address this in the near future. Until we do, be aware of the potential performance impacts of calling `unschedule()`.

Note that when the SmartApp is uninstalled, all scheduled executions are removed - there is no need to call `unschedule()` in the `uninstalled()` method.

114.8.6 Number of scheduled executions limit

The `canSchedule()` (page 902) method returns false if four or more scheduled executions are created.

This currently does not actually impact the ability to create additional schedules, but such a limit may be imposed in the near future. A community post will be made in advance of any such change.

114.9 Examples

Here are some examples in the `SmartThingsPublic` repository that make use of schedules:

- **Once-A-Day** uses `schedule()` to turn switches on and off every day at a specified time.
 - **Turn-It-On** uses `runIn()` to turn a switch off after five minutes.
 - **Left-It-Open** uses `runIn()` to see if a door has been left open for a specified number of minutes.
-

Working With Time

Monitoring the home and triggering Events based on what is detected often entails asking the question: “Is it the right time?” and then based on the answer, perform “Do this, or not,” actions. For example, a SmartApp can turn on a room light when a door is opened but only during certain hours, or wake up the house in the morning at different times based on what day of the week it is.

Time methods can be used in a SmartApp to accomplish such automations. These time methods support a variety of time-related queries such as get the current time or today’s date, know the time zone, or find out if a given moment of time is between a preset time-window.

115.1 Taking action within a time window

A common automation with SmartThings is to turn on a room light when the door is opened between certain hours, and do not turn on the light during other times. The *timeOfDayIsBetween()* (page 943) method comes in handy to set up a SmartApp that accomplishes such an automation.

Refer to the SmartApp code below. First we set up the `preferences()` section with `openCloseSensor`, an open/close sensor that detects when the door is opened, and a `roomLight` that controls the switch to the room light. With the `fromTime` and `toTime` inputs the user will set up the preferred time-window during which the light should be turn on whenever the door is opened.

```
preferences {
    section("Select SmartThings") {
        input "openCloseSensor", "capability.contactSensor", title: "Which door?", required: true, multiple: true
        input "roomLight", "capability.switch", title: "Which room light?", required: true, multiple: true
    }
    section("Turn on between what times?") {
        input "fromTime", "time", title: "From", required: true
        input "toTime", "time", title: "To", required: true
    }
}
```

Next, we begin watching the door by creating the `contactHandler` event handler and have it subscribe to the `contact.open` attribute of the `openCloseSensor` contact sensor. This enables the `contactHandler` event handler to be sensitive only to the `open` Event of the contact sensor, i.e, when the door is opened.

```
def initialize() {
    subscribe(openCloseSensor, "contact.open", contactHandler)
}
```

In the `contactHandler()` implementation below, we ensure our SmartApp performs the following checks:

- Is the door open? No? Then do nothing (in this particular example we do not care if the door is closed).
- If the door is open, then are we within the time-window? No? Then do nothing.
- If the door is open, *and* we are within the time-window, then turn on the room light.

```
def contactHandler(evt) {

    // Door is opened. Now check if the current time is within the visiting hours window
    def between = timeOfDayIsBetween(fromTime, toTime, new Date(), location.timeZone)
    if (between) {
        roomLight.on()
    } else {
        roomLight.off()
    }
}
```

The `timeOfDayIsBetween()` method returns a Boolean `true` or `false` following the logic in the table below.

fromTime	toTime	new Date()	between
12:30	12:32	12:29:59	false
12:30	12:32	12:30:00	true
12:30	12:32	12:30:01	true
12:30	12:32	12:31:59	true
12:30	12:32	12:32:00	true
12:30	12:32	12:32:01	false

115.2 Execute only on certain days

A natural extension to the above automation of taking action within a time window is taking action only within a time window on *selected* days of the week. This can be easily achieved by a slight modification to the above SmartApp.

First we prompt the user to select the preferred days of the week, by adding an enumerated input `days` in the preferences section, as below:

```
preferences {
    section("On Which Days") {
        input "days", "enum", title: "Select Days of the Week", required: true, multiple: true, options: [ ]
    }
}
```

Next, we make modifications to the `contactHandler` event handler so that it checks for the following conditions:

- Is the door open? No? Then do nothing (as in the earlier example, we do not care if the door is closed).
- If the door is open, then is today one of the preferred days-of-the-week?
- If no, then do nothing.
- If yes, i.e., if today is one of the preferred days-of-the-week, then are we within the time-window? No? Then do nothing.
- If yes, then turn on the room light.

```
def contactHandler(evt) {

    // Door is opened. Now check if today is one of the preset days-of-week
    def df = new java.text.SimpleDateFormat("EEEE")
```



```

// Ensure the new date object is set to local time zone
df.setTimeZone(location.timeZone)
def day = df.format(new Date())
//Does the preference input Days, i.e., days-of-week, contain today?
def dayCheck = days.contains(day)
if (dayCheck) {
    def between = timeOfDayIsBetween(fromTime, toTime, new Date(), location.timeZone)
    if (between) {
        roomLight.on()
    } else {
        roomLight.off()
    }
}
}

```

115.3 Working with time zones

Often we may want to set or adjust the SmartApp automation settings while we are traveling, in which case the time zone of the hub may differ from the time zone of the mobile app (our current travel location). For this reason, the code defining the SmartApp should be aware of the time zone of the physical location of the hub.

When working with time-related methods, SmartThings provides ways to handle time zone of both the physical location of the hub and of the mobile app (installed on mobile phone).

For example, `location.getTimeZone()` gives the time zone of the physical location of the hub, whereas invoking `timeZone()` (page 946) method will give the current time zone of the mobile app, i.e., the time zone where mobile phone is currently located.

For a hub that is physically located in Eastern Time Zone in the U.S., and the mobile phone with SmartThings mobile app located in the Pacific Time Zone, the below SmartApp code fragment prints the results shown in the comments:

```

preferences {
    section("What time?") {
        input "myTime", "time", title: "From", required: false
    }
}

...

def contactHandler(evt) {
    // this below outputs "America/New_York", i.e., time zone of hub's physical location
    log.debug "location.getTimeZone() value is: ${location.getTimeZone()}"
    // this below outputs "America/Los_Angeles", the time zone of the mobile app
    log.debug "timeZone() for the preference time input value is: ${timeZone(myTime)}"
}

```

Many time-related methods, such as `timeOfDayIsBetween()` and `timeToday()` require `timeZone` argument to ensure that the correct time zone of the hub is used.

Sunset and Sunrise

SmartApps often need to take some action at or around the local sunrise or sunset time. The SmartThings cloud provides access to this type of rich data, and even generates Events for the Location (if the geofence is set). We can also get access to sunrise and sunset times using a ZIP code.

116.1 Sunrise and sunset Events

Using the sunrise and sunset Events is the preferred (and simpler) way to take some action at (or around) sunrise or sunset. It is required that the Location has set up a geofence.

116.1.1 Taking action at sunrise or sunset

If you wish to have certain actions take place at sunrise or sunset, you can use the *sunrise* and *sunset* Events. These Events will be fired at (gasp!) sunrise and sunset times for the user's Location.

You can subscribe to the Events by passing in the Location (automatically injected into every SmartApp), the event ("sunrise" or "sunset"), and your handler method:

```
def installed() {
    subscribe(location, "sunset", sunsetHandler)
    subscribe(location, "sunrise", sunriseHandler)
}

def sunsetHandler(evt) {
    log.debug "Sun has set!"
    ...
}

def sunriseHandler(evt) {
    log.debug "Sun has risen!"
    ...
}
```

116.1.2 Taking action before or after

If you want to take some action a certain amount of time before or after sunset or sunrise, you can use the "sunriseTime" and "sunsetTime" Events. These Events are fired every day around the time of sunset or sunrise, and their value is the

next sunrise or sunset. You can use this information to calculate an offset so that some action happens a certain amount of time before or after sunrise or sunset.

To use, you can subscribe to the Events by passing the Location, the event (“sunriseTime” or “sunsetTime”), and the handler method.

Consider the following example that turns on lights a specified number of minutes before sunset for the user’s Location:

```
preferences {
    section("Lights") {
        input "switches", "capability.switch", title: "Which lights to turn on?"
        input "offset", "number", title: "Turn on this many minutes before sunset"
    }
}

def installed() {
    initialize()
}

def updated() {
    unsubscribe()
    initialize()
}

def initialize() {
    subscribe(location, "sunsetTime", sunsetTimeHandler)

    //schedule it to run today too
    scheduleTurnOn(location.currentValue("sunsetTime"))
}

def sunsetTimeHandler(evt) {
    //when I find out the sunset time, schedule the lights to turn on with an offset
    scheduleTurnOn(evt.value)
}

def scheduleTurnOn(sunsetString) {
    //get the Date value for the string
    def sunsetTime = Date.parse("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", sunsetString)

    //calculate the offset
    def timeBeforeSunset = new Date(sunsetTime.time - (offset * 60 * 1000))

    log.debug "Scheduling for: $timeBeforeSunset (sunset is $sunsetTime)"

    //schedule this to run one time
    runOnce(timeBeforeSunset, turnOn)
}

def turnOn() {
    log.debug "turning on lights"
    switches.on()
}
```

Because the sunriseTime and sunsetTime Events are fired every day for the *next* sunrise/sunset event, we use runOnce() to schedule one execution. Sunrise and sunset times change, so the next time the Events are fired, we will create another scheduled execution using the runOnce() method for that time.

We want it to run today too, so we use the sunsetTime value of the user’s Location to schedule the lights to turn on today.

Note: If a user changes their Location's geofence, it could change the sunrise and sunset times. You can listen for position change Events and reschedule accordingly: `subscribe(location, "position", locationPositionChangeHandler)`

116.2 Looking up sunrise or sunset directly

SmartApps can use the provided `getSunriseAndSunset()` (page 906) method to get the sunrise and sunset time. You can pass in a ZIP code, which can be useful if the user has not set a geofence for their Location.

The return value is a map in the following form:

```
[sunrise: Date, sunset: Date]
```

```
def initialize() {
    def noParams = getSunriseAndSunset()
    def beverlyHills = getSunriseAndSunset(zipCode: "90210")
    def thirtyMinsBeforeSunset = getSunriseAndSunset(sunsetOffset: "-00:30")

    log.debug "sunrise with no parameters: ${noParams.sunrise}"
    log.debug "sunset with no parameters: ${noParams.sunset}"
    log.debug "sunrise and sunset in 90210: $beverlyHills"
    log.debug "thirty minutes before sunset at current Location: ${thirtyMinsBeforeSunset.sunset}"
}
```

116.3 Polling for sunrise or sunset

You may have seen some SmartApp code that runs a task sometime after midnight (usually in a method called “astroCheck”) and calls a third party weather API to get the sunrise/sunset times. This is strongly discouraged now; it is much more efficient to use Location Events as they do not rely on third party services.

116.4 Examples

You can refer to these example SmartApps in the IDE to see how sunrise and sunset can be used:

- [Smart Nightlight](#)
- [Sunrise/Sunset](#)

You can also refer to the following examples in Github:

- [Sunset Event Example](#)
- [Sunset Offset Example](#)
- [Sunset by ZIP Code Example](#)

App Touch

There are certain cases where we want to perform some action when the user chooses to do so, by clicking on the *Play* icon next to the SmartApp.

For example, a custom voice notification SmartApp might want to play the message when the user presses play.

117.1 Subscribe to app

To enable this feature, you simply subscribe to the app:

```
def initialize() {
    subscribe(app, appHandler)
}

def appHandler(evt) {
    log.debug "app event ${evt.name}:${evt.value} received"
}
```

Simply subscribing to the event will cause the app to display with a play icon in the mobile application:



Your app event handler method can then take the action it needs to in response to the touch event.

Making Synchronous External HTTP Requests

SmartApps or Device Handlers may need to make calls to external web services. There are several APIs available to you to handle making these requests.

The various APIs are named for the underlying HTTP method they will use. `httpGet()` makes an HTTP GET request, for example.

Note: The APIs discussed here are executed synchronously, within a single SmartApp or Device Handler execution.

For information on making asynchronous HTTP requests, check out the *Making Asynchronous External HTTP Requests (Beta)* (page 371) documentation.

118.1 HTTP methods

The following methods are available for making HTTP requests. You can read more about each of them in the *SmartApp* (page 897) API documentation.

These methods execute synchronously, and there is a 10 second timeout limit for the response to be received.

Method	Description
<code>httpDelete()</code> (page 920)	Executes an HTTP DELETE request
<code>httpGet()</code> (page 921)	Executes an HTTP GET request
<code>httpHead()</code> (page 922)	Executes an HTTP HEAD request
<code>httpPost()</code> (page 923)	Executes an HTTP POST request
<code>httpPostJson()</code> (page 923)	Executes an HTTP POST request with JSON Content-Type
<code>httpPutJson()</code> (page 925)	Executes an HTTP PUT request with JSON Content-Type

Here's a simple example of making an HTTP GET request:

```
def params = [
  uri: "http://httpbin.org",
  path: "/get"
]

try {
  httpGet(params) { resp ->
    resp.headers.each {
      log.debug "${it.name} : ${it.value}"
    }
  }
}
```

```
        log.debug "response contentType: ${resp.contentType}"
        log.debug "response data: ${resp.data}"
    }
} catch (e) {
    log.error "something went wrong: $e"
}
```

118.2 Configuring the request

The various APIs for making HTTP requests all accept a map of parameters that define various information about the request:

Parameter	Description
uri	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Request content type and Accept header.
requestContentType	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Note: Specifying a `requestContentType` may override the default behavior of the various http API you are calling. For example, `httpPostJson()` sets the `requestContentType` to `"application/json"` by default.

118.3 Handling the response

The HTTP APIs accept a closure that will be called with the response information from the request.

The closure is passed an instance of a `HttpResponseDecorator`. You can inspect this object to get information about the response.

Here's an example of getting various response information:

```
def params = [
    uri: "http://httpbin.org",
    path: "/get"
]

try {
    httpGet(params) { resp ->
        // iterate all the headers
        // each header has a name and a value
        resp.headers.each {
            log.debug "${it.name} : ${it.value}"
        }

        // get an array of all headers with the specified key
        def theHeaders = resp.getHeaders("Content-Length")
    }
}
```

```

// get the contentType of the response
log.debug "response contentType: ${resp.contentType}"

// get the status code of the response
log.debug "response status code: ${resp.status}"

// get the data from the response body
log.debug "response data: ${resp.data}"
}
} catch (e) {
    log.error "something went wrong: $e"
}
}

```

Tip: Any ‘failed’ response will generate an exception, so you should wrap your calls in a try/catch block.

If the response returns JSON, data will be in a map-like structure that allows you to easily access the response data:

```

def makeJSONWeatherRequest () {
    def params = [
        uri: 'http://api.openweathermap.org/data/2.5/',
        path: 'weather',
        contentType: 'application/json',
        query: [q:'Minneapolis', mode: 'json']
    ]
    try {
        httpGet(params) {resp ->
            log.debug "resp data: ${resp.data}"
            log.debug "humidity: ${resp.data.main.humidity}"
        }
    } catch ( ) {
        log.error "error: $e"
    }
}
}

```

The resp.data from the request above would look like this (indented for readability):

```

resp data: [id:5037649, dt:1432752405, clouds:[all:0],
  coord:[lon:-93.26, lat:44.98], wind:[speed:4.26, deg:233.507],
  cod:200, sys:[message:0.012, sunset:1432777690, sunrise:1432722741,
    country:US],
  name:Minneapolis, base:stations,
  weather:[[id:800, icon:01d, description:Sky is Clear, main:Clear]],
  main:[humidity:73, pressure:993.79, temp_max:298.696, sea_level:1026.82,
    temp_min:298.696, temp:298.696, grnd_level:993.79]]

```

We can easily get the humidity from this data structure as shown above:

```
resp.data.main.humidity
```

118.4 Host and timeout limitations

118.4.1 Host and IP address restrictions

Requests can only be made to publicly accessible hosts. Remember that when executing an HTTP request, the request originates from the SmartThings platform (i.e., the SmartThings cloud), not from the hub itself.

Requests made to local or private hosts are not allowed, and will fail with a `SecurityException`.

118.4.2 Request timeout limit

Requests will timeout after 10 seconds.

Because the request is executed synchronously within a single execution, we encourage you to check out the new (currently beta) *Making Asynchronous External HTTP Requests (Beta)* (page 371) feature.

118.5 Try it out

If you're interested in experimenting with the various HTTP APIs, there are a few tools you can use to try out the APIs without signing up for any API keys.

You can use httpbin.org to test making simple requests. The `httpGet()` example above uses it.

For testing POST requests, you can use [PostCatcher](#). You can generate a target URL and then inspect the contents of the request. Here's an example using `httpPostJson()`:

```
def params = [
    uri: "http://postcatcher.in/catchers/<yourUniquePath>",
    body: [
        param1: [subparam1: "subparam 1 value",
                subparam2: "subparam2 value"],
        param2: "param2 value"
    ]
]

try {
    httpPostJson(params) { resp ->
        resp.headers.each {
            log.debug "${it.name} : ${it.value}"
        }
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.debug "something went wrong: $e"
}
```

118.6 See also

A simple example using `httpGet()` that connects a SmartSense Temp/Humidity Sensor to your Weather Underground personal weather station can be found [here](#).

You can browse some templates in the IDE that use the various HTTP APIs. The Ecobee Service Manager is an example that uses both `httpGet()` and `httpPost()`.

Making Asynchronous External HTTP Requests (Beta)

Beta Feature

The ability to make asynchronous HTTP requests is currently available as a beta development feature.

All beta asynchronous HTTP APIs exist in the `asynhttp_v1 namespace` (page 374). Approximately 30 days after the launch of this beta feature, we will evaluate metrics and your feedback, and make adjustments as necessary.

When released generally, it is likely that the `v1` postfix will be dropped, and a deprecation period will be announced to change existing usages accordingly.

If, for unexpected reasons, usage of asynchronous HTTP requests has negative impacts on the SmartThings platform, SmartThings reserves the right to alter or remove any impacted asynchronous HTTP APIs without notice. This is highly unlikely and every effort will be made to avoid such a scenario.

If you experience issues or have feedback on these asynchronous HTTP APIs, please share them on [this community thread](#).

119.1 Overview

SmartApps and Device Handlers may need to communicate with third party services via HTTP. This can be accomplished using the various HTTP APIs such as `httpGet()`, `httpPost()`, `httpPut()`, etc, as discussed in the *Making Synchronous External HTTP Requests* (page 365) documentation. But, these APIs are synchronous in nature - the currently executing SmartApp or Device Handler waits for the response from the third party. This synchronous execution blocks the current thread executing the SmartApp or Device Handler, and increases the likelihood of hitting the execution timeout.

To address these issues, we're releasing new APIs so SmartApps and Device Handlers can make HTTP requests *asynchronously*. We specify the details of the request, along with the name of a method (that we must implement) to call with the response. SmartThings will then execute the request, and then call the specified request handler method when the response is received.

With asynchronous HTTP requests, we're far less likely to encounter execution timeouts due to a slow third party service.

119.2 Quick example

Let's jump right in and look at an example asynchronous HTTP request. Our example simply makes a GET request to the GitHub API, and logs the response. Don't worry about the details yet, the rest of this documentation will cover it.

```
include 'asynhttp_v1'

def initialize() {
  def params = [
    uri: 'https://api.github.com',
    contentType: 'application/json'
  ]
  def data = [key1: "hello world"]

  asynhttp_v1.get('responseHandlerMethod', params, data)
}

def responseHandlerMethod(response, data) {
  log.debug "got response data: ${response.getData()}"
  log.debug "data map passed to handler method is: $data"
}
```

The first thing you may notice is the `include` directive. This is a new feature in SmartThings that allows various APIs to be grouped together by their functionality. Don't worry too much about it now, it is discussed in detail [below](#) (page 374). For now, just think of it as a way to import a set of APIs that exist in a specific namespace - in this case, "asynhttp_v1".

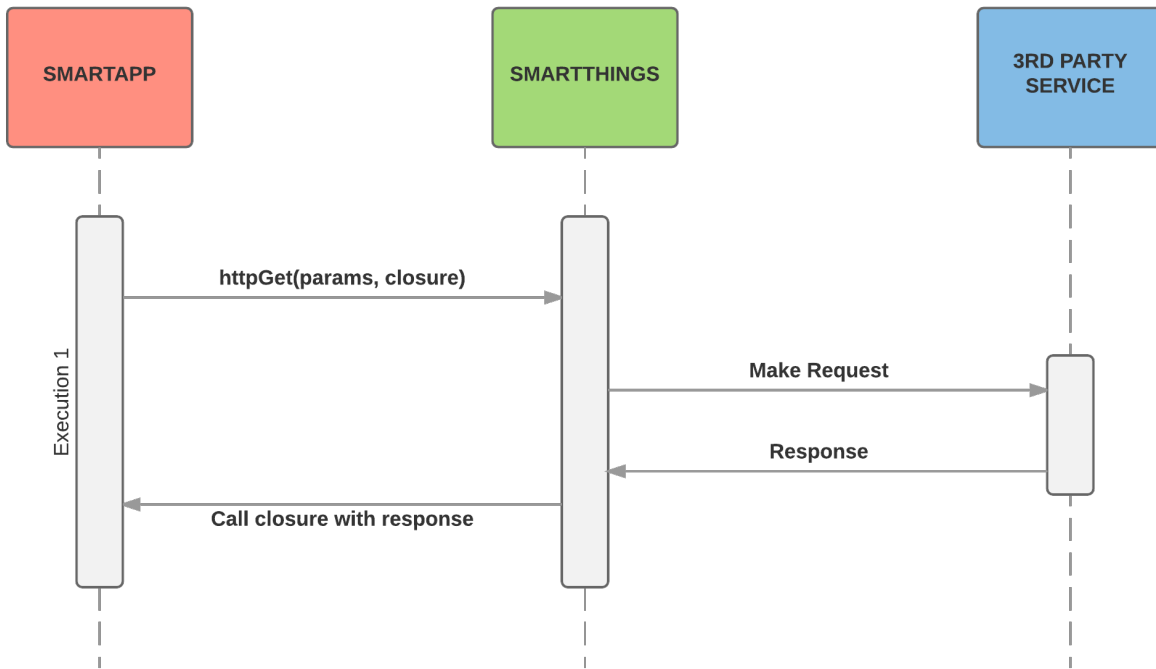
The code to make an asynchronous HTTP request is fairly straightforward. We call `asynhttp_v1.get()` with the name of the method we want to be called with the response, a map of data that is used to build the request, and an optional map of data to pass on to the response handler. The details of the request builder parameters are documented in the [Configuring the request](#) (page 374) section.

We can then define an optional response handler method, which accepts the response of the request, as well as the optional data map we passed to the `get()` method. If none is provided the request will be made in a 'fire-and-forget' mode where the response will be discarded immediately after execution. The details of handling the response are documented in the [Handling the response](#) (page 377) section.

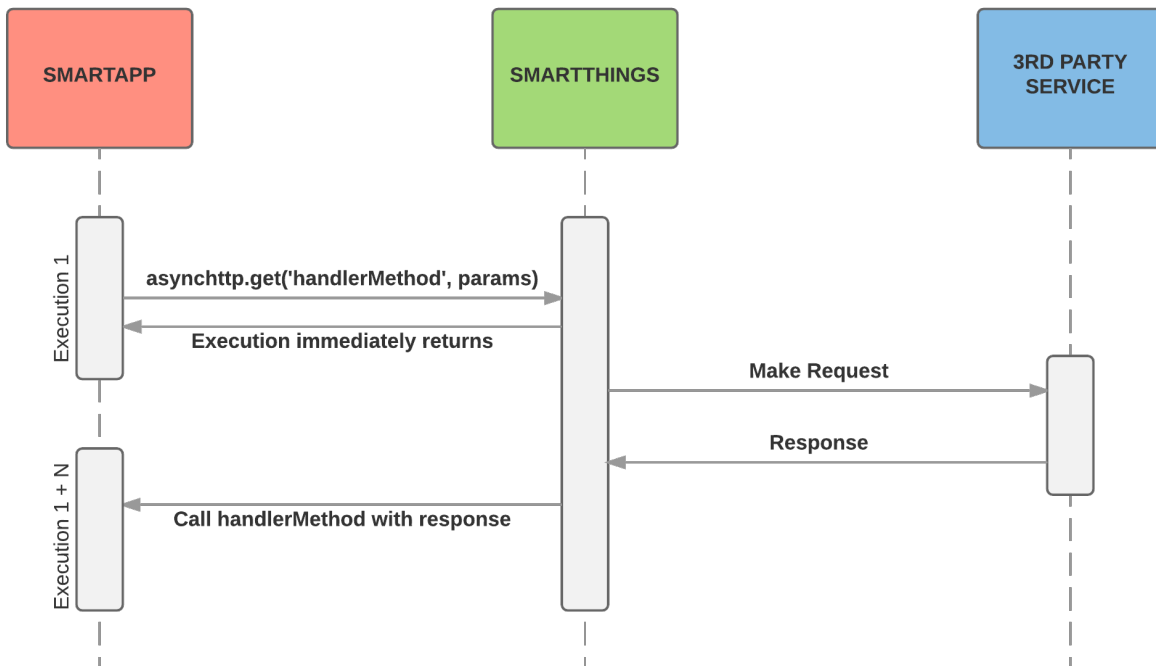
119.3 Synchronous versus asynchronous

The following diagrams illustrate the difference between making synchronous HTTP requests, and using the new asynchronous HTTP APIs.

Synchronous HTTP Request Flow:



Asynchronous HTTP Request Flow:



We can see from the above diagrams that a synchronous HTTP requests makes the requests, waits for the response, then processes the response, all in a single execution.

Asynchronous HTTP requests, on the other hand, handle the response in a *separate execution*. The SmartThings platform makes the request, waits for the response, and then schedules a new SmartApp (or Device Handler) execution

to call the specified response handler with the response.

It is important to note that these executions are not necessarily sequential. Other executions may occur between making the request and receiving the response, either as a result of a scheduled execution or event callbacks. See *When to use asynchronous HTTP requests* (page 382) for more information about using asynchronous versus synchronous HTTP requests.

Asynchronous requests are supported for the GET, POST, PUT, DELETE, HEAD, and PATCH HTTP request methods. A summary of the supported operations is documented *below* (page 381).

119.4 The include Statement

All asynchronous HTTP APIs are namespaced in an object that can be included in the SmartApp or Device Handler using the `include` statement:

```
include 'asynhttp_v1'
```

`asynhttp_v1` is then a reference to an object that the asynchronous HTTP APIs exist on:

```
include 'asynhttp_v1'

def initialize() {
    asynhttp_v1.get([url: 'https://api.github.com'], handler)
}

def handler(response, data) {
    // handle response
}
```

The `include` statement should be placed at the top of the file.

Note: The asynchronous HTTP APIs are the first feature to utilize this feature.

The motivation for this feature is to allow a finer-grained control over the APIs available to SmartApps or Device Handlers, and avoid further polluting the global namespace.

When using `include()`, the SmartThings platform will attempt to find an internally registered API that matches the name provided. If one is found, an instance of the class representing that API will be injected into the SmartApp or Device Handler. If no API is found for the given name, an exception will be thrown and the SmartApp or Device Handler will fail to save.

119.5 Configuring the request

All asynchronous HTTP request methods require, as the first argument, the name of the method to call with the response. We also need to specify some information about the request, such as the URI, an optional path, URL query parameters, HTTP headers, and the content type of the request. We do so by passing a map of parameters. The table below lists the supported keys in the map.

Key	Description
uri (required)	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
request-Content-Type	The value of the <code>Content-Type</code> request header. Defaults to <code>'application/json'</code> .
content-Type	The value of the <code>Accept</code> request header. Defaults to the value of the <code>requestContentType</code> parameter if not specified.
body	The request body to send. Can be a string, or if the <code>requestContentType</code> is <code>"application/json"</code> , a Map or List (will be serialized to JSON). Only valid for PUT, POST, DELETE, and PATCH requests.

119.5.1 URI and path

The `uri` is required for all asynchronous HTTP request methods. If specified, the `path` will be merged with the URI:

```
// uri and path merged to form "https://someapi.com/some/path"
def params = [
    uri: 'https://someapi.com',
    path: '/some/path'
]
```

Note that only publicly accessible (i.e., non-local) addresses can be used when making HTTP requests. See the *Host, timeout, response, and data size limits* (page 381) section below for more information.

119.5.2 Request headers

As you see in the above table, the request headers `Content-Type` and `Accept` will be added to every request. If you need to set other request headers, specify them using the `headers` key in the parameters map:

```
def params = [
    uri: 'https://api.github.com',
    path: '/repos/SmartThingsCommunity/SmartThingsPublic/events',
    headers: ['If-None-Match': 'c873e724d02caa124de0884535c32acb']
]
asynhttp_v1.get('someHandlerMethod', params)
```

As configured above, the request would look like this:

```
GET /repos/SmartThingsCommunity/SmartThingsPublic/events HTTP/1.1
Host: api.github.com
Content-Type: application/json
Accept: application/json
If-None-Match: c873e724d02caa124de0884535c32acb
```

119.5.3 Query parameters

URL query parameters can be added to the request by specifying a map as the value for the `query` key:

```
include 'asynhttp_v1'

def initialize() {
  // search for occurrences of httpGet in the SmartThingsPublic repo
  def params = [
    uri: 'https://api.github.com',
    path: '/search/code',
    query: [q: "httpGet+repo:SmartThingsCommunity/SmartThingsPublic"],
    contentType: 'application/json'
  ]
  asynhttp_v1.get(processResponse, params)
}

def processResponse(response, data) { ... }
```

The request made given the code above would look like this:

```
GET /search/code?q=httpGet+repo:SmartThingsCommunity/SmartThingsPublic HTTP/1.1

Host: api.github.com
Content-Type: application/json
Accept: application/json
```

119.5.4 Request body

HTTP request methods that may have a body can also specify a body in the parameters map. The value of `body` can be a string, or if the `requestContentType` is "application/json", a Map or List (will be serialized to JSON). The `put()` (page 992), `post()` (page 991), `delete()` (page 988), and `patch()` (page 990) methods support the body option.

Here's an example making a POST request using a map for the body:

```
include 'asynhttp_v1'

def initialize() {
  def params = [
    uri: 'https://someapi.com',
    path: '/some/path',
    body: [key1: 'value 1']
  ]
  asynhttp_v1.post(processResponse, params)
}

def processResponse(response, data) { ... }
```

Here's what the request looks like (note that the `Content-Type` and `Accept` headers are "application/json" by default):

```
POST /some/path

Host: someapi.com
Content-Type: application/json
Accept: application/json

{"key1": "value 1"}
```

Here's an example making a PUT request using a string as the body:

```
include 'asynhttp_v1'

def initialize() {
  def params = [
    uri: 'https://someapi.com',
    path: '/some/path',
    body: "<entity><name>test</name></entity>",
    requestContentType: "application/xml"
  ]
  asynhttp_v1.put(processResponse, params)
}

def processResponse(response, data) { ... }
```

And here's the request made by the above example:

```
PUT /some/path

Host: someapi.com
Content-Type: application/xml
Accept: application/xml

<entity><name>test</name></entity>
```

119.6 Handling the response

Once SmartThings executes the request we specified and receives a response from the third party, the request handler method (if specified) will be called (in a new execution of the SmartApp or Device Handler). It will be called with an instance of *AsyncResponse* (page 993), which allows us to get information about the response.

The response handler method must also accept a map of data that may have been specified in the request. This can be useful for passing data between the time we create the request and when the response is received. If no (optional) data was specified when making the request, the request handler method will be called with `null` for the second parameter. We'll discuss this optional data parameter later in this documentation.

The signature of the response handler method should look like:

```
def someResponseHandler(response, data) {}
```

119.6.1 Response status code

We can get the response status code if we need to handle different possible response codes:

```
def responseHandler(response, data) {
  def status = response.status
  switch (status) {
    case 200:
      log.debug "200 returned"
      break
    case 304:
      log.debug "304 returned"
      break
    default:
  }
```

```

        log.warn "no handling for response with status $status"
        break
    }
}

```

119.6.2 Response headers

The `AsyncResponse` object contains all headers from the response as a map of key-value pairs (the return type is `Map<String, String>`):

```

def responseHandler(response, data) {
    def headers = response.headers
    headers.each { header, value ->
        log.debug "$header: $value"
    }
    // can use array notation to get specific header values
    def etagHeader = response.headers['ETag']
}

```

119.6.3 Error responses

Use the `hasError()` (page 997) to check if the response has an error. `hasError()` will return true if any exception occurred during the request.

Any non-2XX response is also considered an error.

You can get any error messages using the `getErrorMessage()` (page 995) method.

```

def responseHandler(response, data) {
    if (response.hasError()) {
        log.debug "response received error: ${response.getErrorMessage()}"
    }
}

```

In the case of an error response, you can also get the response body using `getErrorData()` (page 994), `getErrorJson()` (page 994), or `getErrorXml()` (page 995). Note that these methods will throw an exception if called on a successful response.

```

def responseHandler(response, data) {
    if (response.hasError()) {
        log.debug "error response data: $response.errorData"
        try {
            // exception thrown if json cannot be parsed from response
            log.debug "error response json: $response.errorJson"
        } catch (e) {
            log.warn "error parsing json: $e"
        }
        try {
            // exception thrown if xml cannot be parsed from response
            log.debug "error response xml: $response.errorXml"
        } catch (e) {
            log.warn "error parsing xml: $e"
        }
    }
}

```

119.6.4 JSON responses

If the response from a request is JSON, we can get a fully-formed JSONObject of the response using `getJSON()` (page 996). The example below illustrates getting the JSON response from a GitHub API call to get the occurrences of “httpGet” in the SmartThingsPublic repository.

```
include 'asynhttp_v1'

def initialize() {
    def params = [
        uri: 'https://api.github.com',
        path: '/search/code',
        query: [q: "httpGet+repo:SmartThingsCommunity/SmartThingsPublic"]
    ]
    asynhttp_v1.get(processResponse, params)
}

def processResponse(response, data) {
    if (response.hasError()) {
        log.error "response has error: $response.errorMessage"
    } else {
        def results
        try {
            // json response already parsed into JSONObject object
            results = response.json
        } catch (e) {
            log.error "error parsing json from response: $e"
        }
        if (results) {
            def total = results?.total_count

            log.debug "there are $total occurrences of httpGet in the SmartThingsPublic repo"

            // for each item found, log the name of the file
            results?.items.each { log.debug "httpGet usage found in file $it.name" }
        } else {
            log.debug "did not get json results from response body: $response.data"
        }
    }
}
```

`getJSON()` will throw an Exception if the response body cannot be parsed to JSON, if the request failed to get a response, or if the response status code is not 2XX. See the `getJSON()` (page 996) reference documentation for more information.

119.6.5 XML responses

Handling XML responses is similar to JSON - the XML is parsed into a data structure that we can use:

```
include 'asynhttp_v1'

def initialize() {
    def params = [
        uri: 'https://httpbin.org',
        path: '/xml',
        requestContentType: 'application/xml'
    ]
}
```

```
    asynchttp_v1.get('xmlResultsHandler', params)
}

def xmlResultsHandler(response, data) {
    // results look like:
    // <slideshow title="Sample Slide Show" date="Date of publication" author="Yours Truly">
    //     <slide type="all">
    //         <title>Wake up to WonderWidgets!</title>
    //     </slide>
    // </slideshow>
    if (!response.hasError()) {
        def slideshow
        try {
            slideshow = response.xml
        } catch (e) {
            log.error "error parsing XML from response: $e"
        }
        if (slideshow) {
            log.debug "title: ${slideshow.slide.title.text()}" // -> Wake up to WonderWidgets!
        }
    } else {
        log.error "error making request: ${response.getErrorMessage()}"
    }
}
```

Like `getJSON()`, `getXML()` throws an exception if the results cannot be parsed to XML from the response body. See `getXML()` (page 997) for more information.

119.6.6 Getting the raw response

If we want to get the raw response data, we can do that using `getData()` (page 994).

```
def responseHandler(response, data) {
    log.debug "the raw response data is: $response.data"
}
```

119.7 Passing data to the request handler

Given that the response for an asynchronous HTTP request is processed in a separate SmartApp or Device Handler execution, we may need a way to share data between when we make the request, and when the response handler is called. Rather than store such data in *State* (page 315), we can pass a map of data to any of the asynchronous HTTP methods, and this will be passed along to the response handler:

Note: All response handler methods must accept a second parameter for the data map, even if no data is specified on the request. In that case, the value passed to the response handler will be `null`.

If your response handler does not accept the second parameter, a `MethodMissingException` error will be thrown when the platform tries to call your response handler.

```
include 'asynchttp_v1'
```



```

def initialize() {
    def params = [url: 'https://someapi.com']
    def data = [key1: "value 1", key2: "value 2"]
    asynhttp_v1.get(handler, params, data)
}

def handler(response, data) {
    // logs [key1: "value 1", key2: "value 2"]
    log.debug "data passed to response handler: $data"
}

```

119.8 Available methods

The following methods are available on the `asynhttp_v1` object. The HTTP request method will match the name of the `asynhttp_v1` method—see the reference documentation for more details on each method.

HTTP Verb	Method
GET	<code>asynhttp_v1.get(String callbackMethod, Map params, Map data = null)</code> (page 989)
PUT	<code>asynhttp_v1.put(String callbackMethod, Map params, Map data = null)</code> (page 992)
POST	<code>asynhttp_v1.post(String callbackMethod, Map params, Map data = null)</code> (page 991)
DELETE	<code>asynhttp_v1.delete(String callbackMethod, Map params, Map data = null)</code> (page 988)
PATCH	<code>asynhttp_v1.patch(String callbackMethod, Map params, Map data = null)</code> (page 990)
HEAD	<code>asynhttp_v1.head(String callbackMethod, Map params, Map data = null)</code> (page 990)

119.9 Host, timeout, response, and data size limits

119.9.1 Host and IP address restrictions

Requests can only be made to publicly accessible hosts. Remember that when executing an HTTP request, the request originates from the SmartThings platform (i.e., the SmartThings cloud), not from the hub itself.

Requests made to local or private hosts are not allowed, and will fail with a `SecurityException`.

119.9.2 Request timeout limit

Requests will timeout after 40 seconds. If the request timeout is hit, the response handler will be called and the response will have an error:

```

def responseHandler(response, data) {
    if (response.hasError()) {
        log.error "response has error: $response.errorMessage"
    }
}

```

119.9.3 Response size limit

The current limit is 500,000 characters of response data. This limit will be studied during the beta period, and may be adjusted as necessary.

When the limit is hit, the response body will be empty, but the response status will reflect the actual response status. A warning message will be added to *getWarningMessages()* (page 996) stating that the response size exceeded the limit.

119.9.4 Data size limit

The size of the data map that can be passed to the response handler is limited to 1000 characters when serialized to JSON. If this limit is exceeded, an `IllegalArgumentException` will be thrown when making the request.

119.10 Using asynchronous HTTP in parent-child relationships

When making an asynchronous HTTP request, the associated response handler method will be called on the SmartApp that made the request. This may be obvious, but it is something to keep in mind if you are developing a parent-child relationship SmartApp or Device Handler.

For example, a child SmartApp or Device Handler can call a method on its parent that makes an asynchronous HTTP request, as long as the response handler also exists within the parent.

119.11 When to use asynchronous HTTP requests

Simply put, prefer asynchronous unless it is proven that synchronous is required. Each case needs to be considered on its own, but there are some general cases where synchronous HTTP requests may be required:

- When the response is used in the UI, such as during the OAuth flow during install for cloud-to-cloud device integrations.
- When the response is returned immediately to other APIs, and those APIs cannot be refactored.

The next section discusses some strategies for refactoring synchronous HTTP requests to be asynchronous, and highlights some of the design changes that the asynchronous nature demand.

119.12 Refactoring to asynchronous HTTP requests

119.12.1 Find high-value opportunities

When considering if you should refactor synchronous HTTP requests to be asynchronous, look for high-value opportunities. High-value can be defined as frequent, often scheduled, executions that make HTTP requests.

For example, a SmartApp that executes an HTTP request every five minutes can benefit tremendously from being refactored to use asynchronous HTTP. On the other hand, a single synchronous HTTP request done only during install or in some other low-frequency occurrence, may not benefit as much from being refactored to asynchronous, especially if such a refactoring is costly or risky.

Look for usages of synchronous HTTP requests that occur on schedules or other high-frequency occurrences, and refactor those first.

119.12.2 Refactoring strategies

When refactoring synchronous HTTP requests to be asynchronous, we need to be sure that any code executed after the response has been received is moved to the response callback handler. Consider the following synchronous HTTP example:

```
def initialize() {
  def results = getSomeData()
  log.debug "got results $results"
  doSomethingWithData(results)
}

def getSomeData() {
  def params = [
    uri: 'https://someapi.com',
    path: '/some/path'
  ]
  def results
  httpGet(params) { resp ->
    ...
    results = resp.data
  }
  return results
}

def doSomethingWithData(results) {
  // do something with the results data
}
```

In the example above, the `initialize()` method (and all methods it calls) will execute in a *single execution*. The execution will make the request, wait for the request to return a response, and then parse that response and do something with it.

To change the above example to use the asynchronous HTTP methods, we need to move all code that expects the results into the response handler. We cannot simply update the `getSomeData()` method to use asynchronous HTTP, because the code in `initialize()` following the call to `getSomeData()` assumes that the response has been received.

Below is the updated code to use asynchronous HTTP requests. Because the request is handled asynchronously and the response handler called in another execution, we move the logic that requires the response into the response handler.

```
include 'asynhttp_v1'

def initialize() {
  getSomeData()
}

// execution 1: make the request
def getSomeData() {
  def params = [
    uri: 'https://someapi.com',
    path: '/some/path'
  ]
  asynhttp_v1.get('responseHandler', params)
}
```

```
// execution 1 + n: handle the response
def responseHandler(response, data) {
    def data = response.data
    log.debug "got data: $data"
    doSomethingWithData(data)
}

def doSomethingWithData(results) {
    // do something with the results data
}
```

119.13 Example

A complete SmartApp example illustrating the APIs discussed in this document, along with installation instructions, can be found [here](#).

119.14 Related documentation

- *Making Synchronous External HTTP Requests* (page 365)
- *Async HTTP API (Beta)* (page 987)
- *AsyncResponse (Beta)* (page 993)

Sending Notifications

SmartApps can send notifications, either as a push notification in the mobile app, or as SMS messages to designated recipients. This allows SmartApps to notify people when important Events happen in their home.

120.1 Send notifications with Contact Book

Note: The Contact Book feature is not currently enabled for users. However, using the Contact Book APIs (with the fall-back to non-Contact Book features), will future-proof your SmartApp for when Contact Book is enabled.

See the *Handling disabled Contact Book* (page 387) section for more information.

If a user has added contacts to their Contact Book, SmartApps can prompt a user to select contacts to send notifications to. This allows a user's contacts to be managed independently through the Contact Book, and SmartApps can tap into that feature. This has the advantage that a user does not have to enter in phone numbers for every SmartApp.

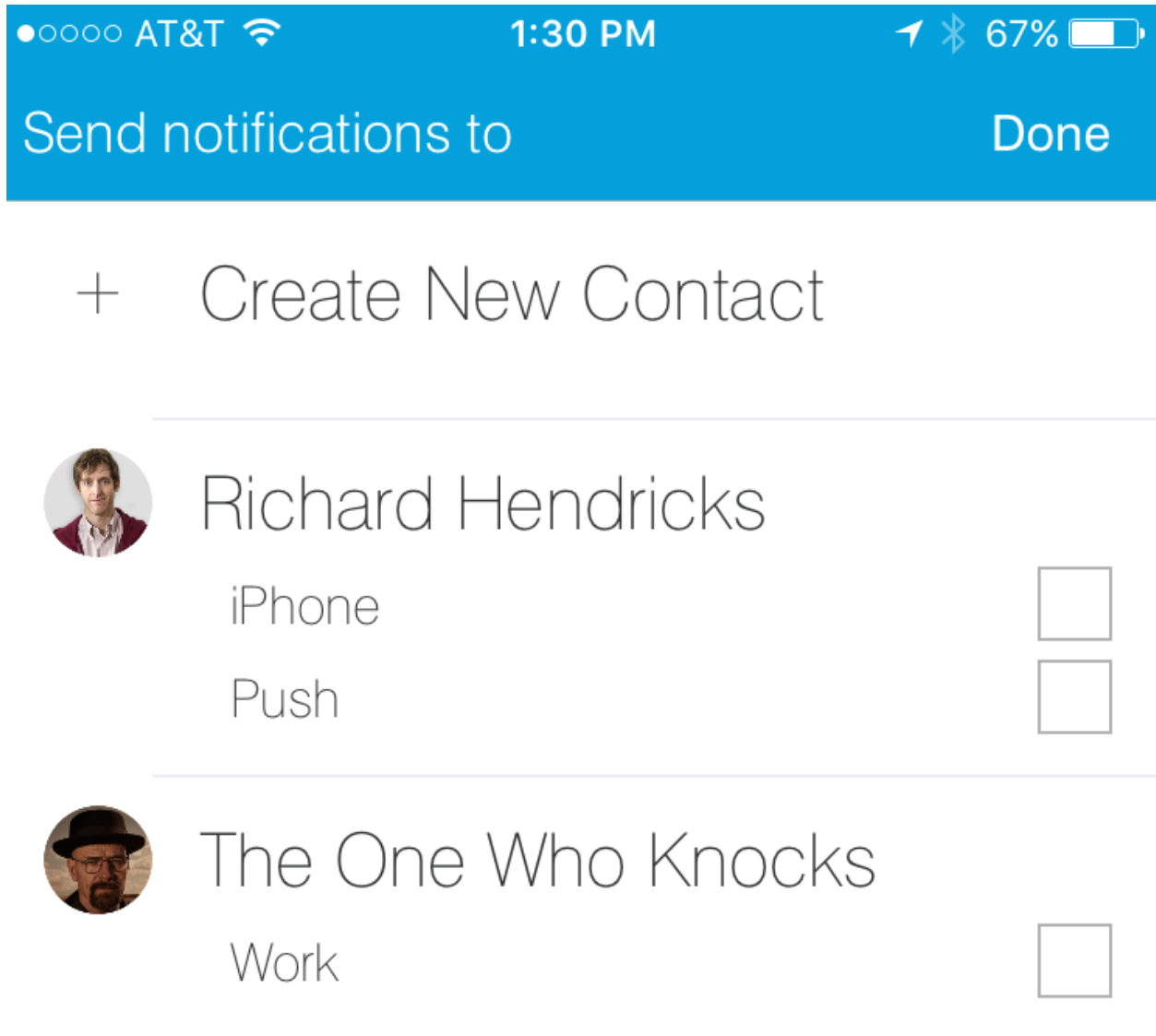
Sending notifications by using the Contact Book feature is the preferred way for sending notifications in a SmartApp.

120.1.1 Selecting Contacts to notify

To allow a user to select from a list of their contacts, use the "contact" input type:

```
preferences {
  section("Send Notifications?") {
    input("recipients", "contact", title: "Send notifications to")
  }
}
```

When the user configures this SmartApp, they can then select which contacts they want to notify, and how they should be notified (SMS or push):



In the example above, the users selected will be stored in a variable named `recipients`. This is just a simple list that we can pass into the `sendNotificationToContacts()` method.

Note: When creating contacts, the user can enter an email address. Emails are *not* currently sent by SmartThings, though they are used to identify SmartThings users, and enable them to receive push notifications.

120.1.2 Send notifications to Contacts

Use the `sendNotificationToContacts()` method to send a notification to the users (and the specified mode of contact) selected.

`sendNotificationToContacts()` accepts three parameters - the message to send, the contacts selected, and an optional map of additional parameters. The valid option for the additional parameters is `[event: false]`, which will suppress the message from appearing in the Notifications feed.

Assuming the "contact" input named "recipients" above, you would use:

```
sendNotificationToContacts("something you care about!", recipients)
```

If you don't want the message to appear in the Notifications feed, specify event : false:

```
sendNotificationToContacts("something you care about!", recipients, [event: false])
```

120.1.3 Handling disabled Contact Book

A user may not have created any contacts, and SmartApps should be written to handle this.

The "contact" input element takes an optional closure, where you can define additional input elements that will be displayed if the user has no contacts. If the user has contacts, these input elements won't be seen when installing or configuring the SmartApp.

Modifying our preferences definition from above, to handle the case of a user having no contacts, would look like:

```
preferences {
    section("Send Notifications?") {
        input("recipients", "contact", title: "Send notifications to") {
            input("phone", "phone", title: "Warn with text message (optional)",
                description: "Phone Number", required: false)
        }
    }
}
```

If the user configuring this SmartApp does have contacts defined, they will only see the input to select from those contacts. If they don't have any contacts defined, they will see the input to enter a phone number.

When attempting to send notifications, we should also check to see if the user has enabled the Contact Book and selected contacts. You can check the `contactBookEnabled` property on `location` to find out if Contact Book has been enabled. It's a good idea to also check if any contacts have been selected.

```
// check that Contact Book is enabled and recipients selected
if (location.contactBookEnabled && recipients) {
    sendNotificationToContacts("your message here", recipients)
} else if (phone) { // check that the user did select a phone number
    sendSms(phone, "your message here")
}
```

120.1.4 Complete example

The example SmartApp below sends a notification to selected contacts when a door opens. If the user has no contacts, they can enter in a number to receive an SMS notification.

```
definition(
    name: "Contact Book Example",
    namespace: "smarththings",
    author: "SmartThings",
    description: "Example using Contact Book",
    category: "My Apps",
    iconUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
    iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",
    iconX3Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png")
```

```
preferences {
    section("Which Door?") {
        input "door", "capability.contactSensor", required: true,
            title: "Which Door?"
    }

    section("Send Notifications?") {
        input("recipients", "contact", title: "Send notifications to") {
            input "phone", "phone", title: "Warn with text message (optional)",
                description: "Phone Number", required: false
        }
    }
}

def installed() {
    initialize()
}

def updated() {
    initialize()
}

def initialize() {
    subscribe(door, "contact.open", doorOpenHandler)
}

def doorOpenHandler(evt) {
    log.debug "recipients configured: $recipients"

    def message = "The ${door.displayName} is open!"
    if (location.contactBookEnabled && recipients) {
        log.debug "Contact Book enabled!"
        sendNotificationToContacts(message, recipients)
    } else {
        log.debug "Contact Book not enabled"
        if (phone) {
            sendSms(phone, message)
        }
    }
}
```

Note: The rest of this guide discusses alternative ways to send notifications (push, SMS, Notifications Feed). SmartApps should use Contact Book, and use the methods described below as a precaution in case the user does not have Contact Book enabled.

120.2 Send push notifications

To send a push notification through the SmartThings mobile app, you can use the `sendPush()` or `sendPushMessage()` methods. Both methods simply take the message to display. `sendPush()` will display the message in the Notifications feed; `sendPushMessage()` will not.

A simple example below shows (optionally) sending a push message when a door opens:


```

preferences {
    section("Which door?") {
        input "door", "capability.contactSensor", required: true,
            title: "Which door?"
    }
    section("Send Push Notification?") {
        input "sendPush", "bool", required: false,
            title: "Send Push Notification when Opened?"
    }
}

def installed() {
    initialize()
}

def updated() {
    initialize()
}

def initialize() {
    subscribe(door, "contact.open", doorOpenHandler)
}

def doorOpenHandler(evt) {
    if (sendPush) {
        sendPush("The ${door.displayName} is open!")
    }
}

```

Push notifications will be sent to all users with the SmartThings mobile app installed, for the account the SmartApp is installed into.

120.3 Send SMS notifications

In addition to sending push notifications through the SmartThings mobile app, you can also send SMS messages to specified numbers using the `sendSms()` and `sendSmsMessage()` methods.

Both methods take a phone number (as a string) and a message to send. The message can be no longer than 140 characters. `sendSms()` will display the message in the Notifications feed; `sendSmsMessage()` will not.

Extending the example above, let's add the ability for a user to (optionally) send an SMS message to a specified number:

```

preferences {
    section("Which door?") {
        input "door", "capability.contactSensor", required: true,
            title: "Which door?"
    }
    section("Send Push Notification?") {
        input "sendPush", "bool", required: false,
            title: "Send Push Notification when Opened?"
    }
    section("Send a text message to this number (optional)") {
        input "phone", "phone", required: false
    }
}

```

```
}  
  
def installed() {  
    initialize()  
}  
  
def updated() {  
    initialize()  
}  
  
def initialize() {  
    subscribe(door, "contact.open", doorOpenHandler)  
}  
  
def doorOpenHandler(evt) {  
    def message = "The ${door.displayName} is open!"  
    if (sendPush) {  
        sendPush(message)  
    }  
    if (phone) {  
        sendSms(phone, message)  
    }  
}
```

SMS notifications will be sent from the number 844647 (“THINGS”).

120.4 Send both push and SMS notifications

The `sendNotification()` method allows you to send both push and/or SMS messages, in one convenient method call. It can also optionally display the message in the Notifications feed.

`sendNotification()` takes a message parameter, and a map of options that control how the message should be sent, if the message should be displayed in the Notifications feed, and a phone number to send an SMS to (if specified):

```
// sends a push notification, and displays it in the Notifications feed  
sendNotification("test notification - no params")  
  
// same as above, but explicitly specifies the push method (default is push)  
sendNotification("test notification - push", [method: "push"])  
  
// sends an SMS notification, and displays it in the Notifications feed  
sendNotification("test notification - sms", [method: "phone", phone: "1234567890"])  
  
// Sends a push and SMS message, and displays it in the Notifications feed  
sendNotification("test notification - both", [method: "both", phone: "1234567890"])  
  
// Sends a push message, and does not display it in the Notifications feed  
sendNotification("test notification - no event", [event: false])
```

120.5 Only display message in the notifications feed

Use the `sendNotificationEvent()` method to display a message in the Notifications feed, without sending a push notification or SMS message:

```
sendNotificationEvent("Your home talks!")
```

120.6 Examples

Several examples exist in the SmartApp templates that send notifications. Here are a few you can look at to learn more:

- [Notify Me When](#) sends push or text messages in response to a variety of Events.
 - [Presence Change Push](#) and [Presence Change Text](#) send notifications when people arrive or depart.
-

120.7 Related API documentation

- [sendNotificationToContacts\(\)](#) (page 938)
- [getContactBookEnabled\(\)](#) (page 1036)
- [sendPush\(\)](#) (page 939)
- [sendPushMessage\(\)](#) (page 939)
- [sendSms\(\)](#) (page 940)
- [sendSmsMessage\(\)](#) (page 940)
- [sendNotification\(\)](#) (page 938)
- [sendNotificationEvent\(\)](#) (page 938)

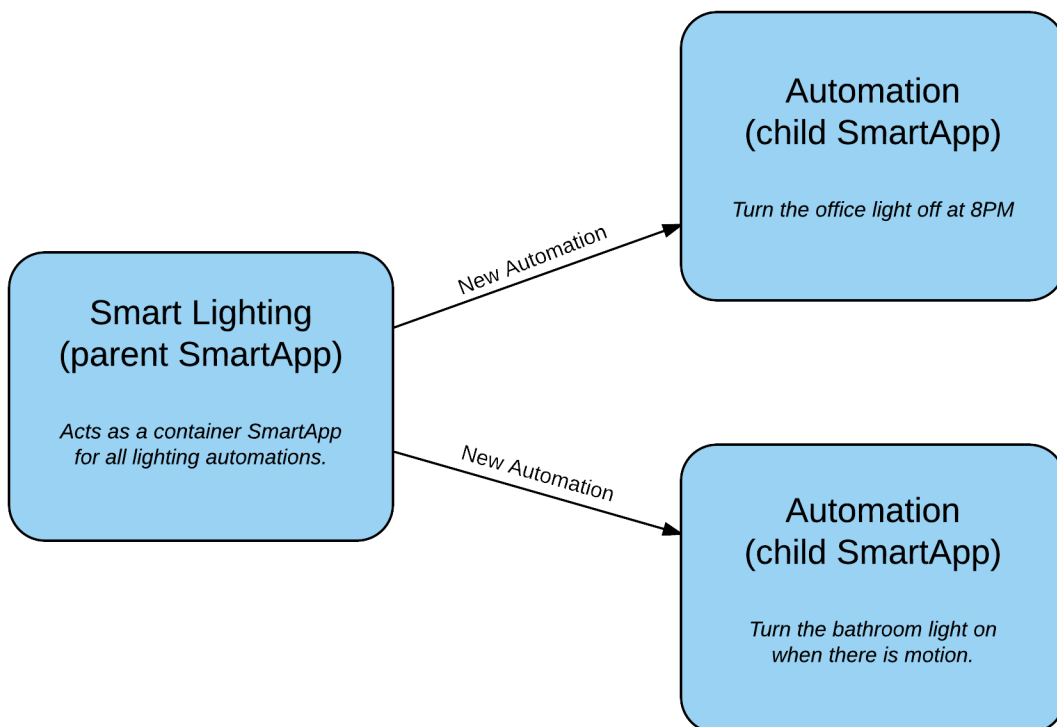
Parent-Child SmartApps

SmartApps can have child SmartApps. This is often useful when you want to provide multiple automations that act independently on separate devices. This will consolidate multiple separate automations under one parent.

121.1 Overview

Smart Lighting is an example of a parent-child SmartApp. When you install Smart Lighting, you are installing one parent SmartApp (Smart Lighting), and each unique lighting automation you create is actually a new instance of a child SmartApp. This child SmartApp is what actually controls each lighting automation.

The diagram below illustrates this relationship:



The relationship between a parent SmartApp and its children is a one-to-many relationship. A SmartApp may have many children, and those children can also have children. A child SmartApp can have only one parent.

121.2 The parent SmartApp

To define that a SmartApp is a parent to other SmartApps, use the `app` input element inside the `preferences`. The `app` input allows the user to install and edit child SmartApp instances, and establishes the relationship between parent and child.

```
preferences {
    page(name: "mainPage", title: "Child Apps", install: true, uninstall: true) {
        section {
            app(name: "childApps", appName: "Child App", namespace: "mynamespace", title: "New Child
        }
    }
}
```

All child SmartApps installed via the `app` input will then be listed in the parent SmartApp preferences page, and the user can then edit or remove those instances.

The options for the `app` input are:

Option	Description
<code>name</code>	The name of the input. Serves as the identifier for this input element.
<code>appName</code>	The name of the child SmartApp, as defined in the definition metadata of the child.
<code>namespace</code>	The namespace of the child SmartApp, as defined in the definition metadata of the child.
<code>title</code>	The title of the button the user can press to install a new instance of this child SmartApp.
<code>multiple</code>	If <code>true</code> , the user can install multiple child SmartApps. If <code>false</code> , only one may be installed. Defaults to <code>false</code> .

121.3 The child SmartApp

In the SmartApp you wish to serve as the child, specify the `parent` option in the child SmartApp's definition, in the form of `"namespace": "Parent App Name"`:

```
definition(
    ...
    parent: "yourNameSpace:Parent App Name",
    ...
)
```

Note: When you save the child SmartApp, the platform will validate that a parent SmartApp with the namespace and name as specified in the `parent` option exists. If it does not, an error will be raised.

Either make sure your parent SmartApp has been saved first, or come back and add the `parent` option after your parent SmartApp has been saved.

121.4 Communicating between parent and children

Parents and children may need to talk to each other. In the parent SmartApp, you can get the child SmartApp using the `getChildApps()` method:

```
def children = getChildApps()
log.debug "$children.size() child apps installed"
children.each { child ->
    log.debug "Child app id: $child.id"
}
```

You can then call methods directly on the child SmartApp:

```
// assumes the child SmartApp has method foo() defined
child.foo()
```

You can also use the `findChildAppByName()` method to find a specific child SmartApp by its name:

```
def theChild = findChildAppByName("My Child App")
```

Children can communicate with their parent by using the `parent` property in the Child SmartApp:

```
// assumes the parent SmartApp has a method bar() defined:
parent.bar()
```

121.5 Preventing more than one parent instance

If you want to prevent users from installing more than one parent SmartApp in their Location, you can specify `singleInstance: true` in the definition:

```
definition(
    ...
    singleInstance: true
    ...
)
```

With `singleInstance: true`, when a user tries to install a parent SmartApp that has already been installed, they will be taken to the existing installation. From there, they can configure existing child SmartApps or add new ones. This avoids having multiple instances of parent SmartApp, when only one is necessary.

121.6 Example

Below is a simple example illustrating how a parent SmartApp (“Simple Lighting”) can be created to allow multiple child SmartApps (“Simple Automations”).

Here is the parent SmartApp:

```
definition(
    name: "Simple Lighting",
    namespace: "mynamespace/parent",
    author: "Your Name",
```

```

description: "An example of parent/child SmartApps (this is the parent).",
category: "My Apps",
iconUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",
iconX3Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png")

preferences {
    // The parent app preferences are pretty simple: just use the app input for the child app.
    page(name: "mainPage", title: "Simple Automations", install: true, uninstall: true, submitOnChange: true)
    section {
        app(name: "simpleAutomation", appName: "Simple Automation", namespace: "mynamespace/automations")
    }
}

def installed() {
    log.debug "Installed with settings: ${settings}"
    initialize()
}

def updated() {
    log.debug "Updated with settings: ${settings}"
    unsubscribe()
    initialize()
}

def initialize() {
    // nothing needed here, since the child apps will handle preferences/subscriptions
    // this just logs some messages for demo/information purposes
    log.debug "there are ${childApps.size()} child smartapps"
    childApps.each {child ->
        log.debug "child app: ${child.label}"
    }
}

```

Here's the child SmartApp:

```

definition(
    name: "Simple Automation",
    namespace: "mynamespace/automations",
    author: "Your Name",
    description: "A simple app to control basic lighting automations. This is a child app.",
    category: "My Apps",

    // the parent option allows you to specify the parent app in the form <namespace>/<app name>
    parent: "mynamespace/parent:Simple Lighting",
    iconUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
    iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",
    iconX3Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png")

preferences {
    page name: "mainPage", title: "Automate Lights & Switches", install: false, uninstall: true, nextScreen: "namePage"
    page name: "namePage", title: "Automate Lights & Switches", install: true, uninstall: true
}

def installed() {
    log.debug "Installed with settings: ${settings}"
}

```



```

    initialize()
}

def updated() {
    log.debug "Updated with settings: ${settings}"
    unschedule()
    initialize()
}

def initialize() {
    // if the user did not override the label, set the label to the default
    if (!overrideLabel) {
        app.updateLabel(defaultLabel())
    }
    // schedule the turn on and turn off handlers
    schedule(turnOnTime, turnOnHandler)
    schedule(turnOffTime, turnOffHandler)
}

// main page to select lights, the action, and turn on/off times
def mainPage() {
    dynamicPage(name: "mainPage") {
        section {
            lightInputs()
            actionInputs()
        }
        timeInputs()
    }
}

// page for allowing the user to give the automation a custom name
def namePage() {
    if (!overrideLabel) {
        // if the user selects to not change the label, give a default label
        def l = defaultLabel()
        log.debug "will set default label of $l"
        app.updateLabel(l)
    }
    dynamicPage(name: "namePage") {
        if (overrideLabel) {
            section("Automation name") {
                label title: "Enter custom name", defaultValue: app.label, required: false
            }
        } else {
            section("Automation name") {
                paragraph app.label
            }
        }
        section {
            input "overrideLabel", "bool", title: "Edit automation name", defaultValue: "false", required: true
        }
    }
}

// inputs to select the lights
def lightInputs() {
    input "lights", "capability.switch", title: "Which lights do you want to control?", multiple: true
}

```

```

// inputs to control what to do with the lights (turn on, turn on and set color, turn on
// and set level)
def actionInputs() {
    if (lights) {
        input "action", "enum", title: "What do you want to do?", options: actionOptions(), required: true
        if (action == "color") {
            input "color", "enum", title: "Color", required: true, multiple: false, options: [
                ["Soft White": "Soft White - Default"],
                ["White": "White - Concentrate"],
                ["Daylight": "Daylight - Energize"],
                ["Warm White": "Warm White - Relax"],
                ["Red", "Green", "Blue", "Yellow", "Orange", "Purple", "Pink"]
            ]
        }
        if (action == "level" || action == "color") {
            input "level", "enum", title: "Dimmer Level", options: [[10:"10%"], [20:"20%"], [30:"30%"], [40:"40%"], [50:"50%"], [60:"60%"], [70:"70%"], [80:"80%"], [90:"90%"], [100:"100%"]]
        }
    }
}

// utility method to get a map of available actions for the selected switches
def actionMap() {
    def map = [on: "Turn On", off: "Turn Off"]
    if (lights.find{it.hasCommand('setLevel')} != null) {
        map.level = "Turn On & Set Level"
    }
    if (lights.find{it.hasCommand('setColor')} != null) {
        map.color = "Turn On & Set Color"
    }
    map
}

// utility method to collect the action map entries into maps for the input
def actionOptions() {
    actionMap().collect{[it.key]: it.value}
}

// inputs for selecting on and off time
def timeInputs() {
    if (settings.action) {
        section {
            input "turnOnTime", "time", title: "Time to turn lights on", required: true
            input "turnOffTime", "time", title: "Time to turn lights off", required: true
        }
    }
}

// a method that will set the default label of the automation.
// It uses the lights selected and action to create the automation label
def defaultLabel() {
    def lightsLabel = settings.lights.size() == 1 ? lights[0].displayName : lights[0].displayName + " and " + lights[1].displayName

    if (action == "color") {
        "Turn on and set color of $lightsLabel"
    } else if (action == "level") {
        "Turn on and set level of $lightsLabel"
    } else {
        "Turn $action $lightsLabel"
    }
}

```

```
    }  
  }  
  
  // the handler method that turns the lights on and sets level and color if specified  
  def turnOnHandler() {  
    // switch on the selected action  
    switch(action) {  
      case "level":  
        lights.each {  
          // check to ensure the switch does have the setLevel command  
          if (it.hasCommand('setLevel')) {  
            log.debug("Not So Smart Lighting: $it.displayName setLevel($level)")  
            it.setLevel(level as Integer)  
          }  
          it.on()  
        }  
        break  
      case "on":  
        log.debug "on()"  
        lights.on()  
        break  
      case "color":  
        setColor()  
        break  
    }  
  }  
}  
  
// set the color and level as specified, if the user selected to set color.  
def setColor() {  
  
  def hueColor = 0  
  def saturation = 100  
  
  switch(color) {  
    case "White":  
      hueColor = 52  
      saturation = 19  
      break;  
    case "Daylight":  
      hueColor = 53  
      saturation = 91  
      break;  
    case "Soft White":  
      hueColor = 23  
      saturation = 56  
      break;  
    case "Warm White":  
      hueColor = 20  
      saturation = 80  
      break;  
    case "Blue":  
      hueColor = 70  
      break;  
    case "Green":  
      hueColor = 39  
      break;  
    case "Yellow":  
      hueColor = 25
```

```
        break;
    case "Orange":
        hueColor = 10
        break;
    case "Purple":
        hueColor = 75
        break;
    case "Pink":
        hueColor = 83
        break;
    case "Red":
        hueColor = 100
        break;
}

def value = [switch: "on", hue: hueColor, saturation: saturation, level: level as Integer ?: 100]
log.debug "color = $value"

lights.each {
    if (it.hasCommand('setColor')) {
        log.debug "$it.displayName, setColor($value)"
        it.setColor(value)
    } else if (it.hasCommand('setLevel')) {
        log.debug "$it.displayName, setLevel($value)"
        it.setLevel(level as Integer ?: 100)
    } else {
        log.debug "$it.displayName, on()"
        it.on()
    }
}

// simple turn off lights handler
def turnOffHandler() {
    lights.off()
}
```

To try it out, create the parent and child SmartApp with the code as shown above, and publish the parent SmartApp for yourself (you don't need to publish the child SmartApp, since it will be discovered by the parent and you don't want to install it individually from the Marketplace). Then, go to the Marketplace and install "Simple Lighting" in "My Apps". You can then add multiple automations, with each automation being an instance of the child SmartApp ("Simple Automation").

121.7 Tips and best practices

- Think carefully about creating more than one level of parent-to-child relationships, as it may negatively impact usability and create unneeded complications.
 - Sharing `state` or `atomicState` between parent and child SmartApps is not currently supported.
 - The number of children a SmartApp may have is capped as documented in the *Parent-child relationship limit* (page 617).
-

121.8 Summary

Parent-child relationships can be useful when you want to provide multiple automations that act independently on separate devices. A parent SmartApp may have many children; a child SmartApp has only one parent.

To create a parent-child relationship, the SmartApp that is to be the parent should use the `app` input type to specify what app can be a child. The child SmartApp should specify the `parent` option in its definition to specify what SmartApp should serve as the parent.

A parent SmartApp can get its children by using the `getChildApps()`, or `findChildAppByName()` if you know the name of the app you are looking for. Children can get a reference to their parent through the `parent` property.

Example: Bon Voyage

To help illustrate some of the important concepts in writing a SmartApp, let's walk through an example.

122.1 Bon Voyage

Our example SmartApp is fairly simple - it will monitor a set of presence detectors, and trigger a Mode change when everyone has left.

To accomplish this, our app will need to do the following:

- Gather the necessary input from the user, including which sensors to monitor, what Mode to trigger, and other app preferences.
- Subscribe to the appropriate Events, and take action when they are triggered.

Let's begin with configuring the preferences.

122.2 SmartApp preferences

To configure the Bon Voyage app, we will want to gather the following information from the user:

- Which sensors to monitor
- The Mode to trigger when everyone is away
- A false alarm threshold
- Who should be notified, and how

Each of these inputs corresponds into a preferences section:

```
preferences {
  section("When all of these people leave home") {
    input "people", "capability.presenceSensor", multiple: true
  }
  section("Change to this mode") {
    input "newMode", "mode", title: "Mode?"
  }
  section("False alarm threshold (defaults to 10 min)") {
```

```

        input "falseAlarmThreshold", "decimal", title: "Number of minutes", required: false
    }
    section( "Notifications" ) {
        input("recipients", "contact", title: "Send notifications to", required: false) {
            input "sendPushMessage", "enum", title: "Send a push notification?", options: ["Yes", "No"]
            input "phone", "phone", title: "Send a Text Message?", required: false
        }
    }
}

```

Let's look at each section in a bit more detail.

The *When all of these people leave home* section allows the user to configure what sensors to use for this app. The user will see a section with the main title “When all of these people leave home.” A drop down will be populated with all the devices that have the `presenceSensor` capability (`capability.presenceSensor`) for them to select the sensor(s) they'd like to use. `multiple: true` allows them to add as many sensors as they'd like. Their choice(s) are then stored in a variable named `people`.

The *Change to this mode* section allows the user to specify what Mode should be triggered when everyone is away. The input type of `mode` is used, so a drop down will be populated with all the modes the user has set up. The title property is used to show the title “Mode?” above the field. The selection is stored in the variable named `newMode`.

The section *False alarm threshold (defaults to 10 min)* allows the user to specify a false alarm threshold. These types of thresholds are common in our SmartApps. A section is shown titled “False alarm threshold (defaults to 10 min)”. The input field type of `decimal` is used, to allow the user to input a numeric value that represents minutes. The title “Number of minutes” is specified, and we set the `required` property to `false`. By default, all fields are required, so you must explicitly state if it is not required. We store the user's input in the variable named `falseAlarmThreshold` for later use.

Finally, a section is shown labeled as “Notifications”. This is where the user can configure how they want to be notified when everyone is away. This input is a little different; it uses the special input type `contact`. You can read more about sending notifications in a SmartApp in the *Sending Notifications* (page 385) section.

122.3 Monitor and react

Now that we have gathered the input we need from the user, we need to listen to the appropriate Events, and take action when they are triggered.

We do this through the required `installed()` method:

```

def installed() {
    log.debug "Installed with settings: ${settings}"
    log.debug "Current mode = ${location.mode}, people = ${people.collect{it.label + ': ' + it.currentPresence}}
    subscribe(people, "presence", presence)
}

```

Upon installation, we want to keep track of the status of our people. We use the `subscribe()` method to listen to the `presence` attribute of the predefined group of presence sensors, `people`. When the presence status changes of any of our people, the method `presence` (the last parameter above) will be called.

(Also note the `log` statements. We won't discuss `log` statements in detail, but providing thorough logging is a habit you will want to get into as a SmartApps developer. It is invaluable when trying to debug or troubleshoot your app!)

Let's define our `presence` method.


```

def presence(evt) {
    log.debug "evt.name: $evt.value"
    if (evt.value == "not present") {
        if (location.mode != newMode) {
            log.debug "checking if everyone is away"
            if (everyoneIsAway()) {
                log.debug "starting sequence"
                runIn(findFalseAlarmThreshold() * 60, "takeAction", [overwrite: false])
            }
        }
        else {
            log.debug "mode is the same, not evaluating"
        }
    }
    else {
        log.debug "present; doing nothing"
    }
}

// returns true if all configured sensors are not present,
// false otherwise.
private everyoneIsAway() {
    def result = true
    // iterate over our people variable that we defined
    // in the preferences method
    for (person in people) {
        if (person.currentPresence == "present") {
            // someone is present, so set our our result
            // variable to false and terminate the loop.
            result = false
            break
        }
    }
    log.debug "everyoneIsAway: $result"
    return result
}

// gets the false alarm threshold, in minutes. Defaults to
// 10 minutes if the preference is not defined.
private findFalseAlarmThreshold() {
    // In Groovy, the return statement is implied, and not required.
    // We check to see if the variable we set in the preferences
    // is defined and non-empty, and if it is, return it. Otherwise,
    // return our default value of 10
    (falseAlarmThreshold != null && falseAlarmThreshold != "") ? falseAlarmThreshold : 10
}

```

Let's break that down a bit.

The first thing we need to do is see what event was triggered. We do this by inspecting the `evt` variable that is passed to our event handler. The presence capability can be either "present" or "not present".

Next, we check that the current Mode isn't already set to the Mode we want to trigger. If we're already in our desired Mode, there's nothing else for us to do!

Now it starts to get fun!

We have defined two helper methods above: `everyoneIsAway()` and `findFalseAlarmThreshold()`.

`everyoneIsAway()` returns true if all configured sensors are not present, and false otherwise. It iterates over all

the sensors configured and stored in the `people` variable, and inspects the `currentPresence` property. If the `currentPresence` is "present", we set the result to false, and terminate the loop. We then return the value of the result variable.

`findFalseAlarmThreshold()` gets the false alarm threshold, in minutes, as configured by the user. If the threshold preference has not been set, it returns 10 minutes as the default.

If everyone is away, we call the built-in `runIn()` (page 928) method, which runs the method `takeAction()` in a specified amount of time (we'll define that method shortly). We use `findFalseAlarmThreshold()` multiplied by 60 to convert minutes to seconds, which is what the `runIn()` method requires. We specify `overwrite: false` so it won't overwrite previously scheduled `takeAction()` calls. In the context of this SmartApp, it means that if one user leaves, and then another user leaves within the false alarm threshold time, `takeAction()` will still be called twice. By default, `overwrite` is true, so any previously scheduled `takeAction()` calls would be canceled and replaced by your current call.

Now we need to define our `takeAction()` method:

```
def takeAction() {
  if (everyoneIsAway()) {
    def threshold = 1000 * 60 * findFalseAlarmThreshold() - 1000
    def awayLongEnough = people.findAll { person ->
      def presenceState = person.currentState("presence")
      def elapsed = now() - presenceState.rawDateCreated.time
      elapsed >= threshold
    }
    log.debug "Found ${awayLongEnough.size()} out of ${people.size()} person(s) who were away long enough"
    if (awayLongEnough.size() == people.size()) {
      //def message = "${app.label} changed your mode to '${newMode}' because everyone left home"
      def message = "SmartThings changed your mode to '${newMode}' because everyone left home"
      log.info message
      send(message)
      setLocationMode(newMode)
    } else {
      log.debug "not everyone has been away long enough; doing nothing"
    }
  } else {
    log.debug "not everyone is away; doing nothing"
  }
}

private send(msg) {
  if ( sendPushMessage != "No" ) {
    log.debug( "sending push message" )
    sendPush( msg )
  }

  if ( phone ) {
    log.debug( "sending text message" )
    sendSms( phone, msg )
  }

  log.debug msg
}
```

There's a lot going on here, so we'll look at some of the more interesting parts.

The first thing we do is check again if everyone is away. This is necessary since something may have changed since it was already called, because of the `falseAlarmThreshold()`.

If everyone is away, we need to find out how many people have been away for long enough, using our `false`

alarm threshold. We create a variable, `awayLongEnough` and set it through the Groovy `findAll()` method. The `findAll()` method returns a subset of the collection based on the logic of the passed-in closure. For each person, we use the `currentState()` (page 1006) method available to us, and use that to get the time elapsed since the event was triggered. If the time elapsed since this event exceeds our threshold, we add it to the `awayLongEnough` collection by returning `true` in our closure (note that we could omit the “return” keyword, as it is implied in Groovy).

For more information about the `findAll()` method, or how Groovy utilizes closures, consult the Groovy documentation at <http://www.groovy-lang.org/documentation.html>

If the number of people away long enough equals the total number of people configured for this app, we send a message (we’ll look at that method next), and then call the `setLocationMode()` (page 940) method with the desired Mode. This is what will cause a Mode change.

The `send()` method takes a String parameter, `msg`, which is the message to send. This is where our app sends a notification to any of the contacts the user has specified.

Finally, we need to write our `updated()` method, which is called whenever the user changes any of their preferences. When this method is called, we need to call the `unsubscribe()` method, and then `subscribe()`, to effectively reset our app.

```
def updated() {
    log.debug "Updated with settings: ${settings}"
    log.debug "Current mode = ${location.mode}, people = ${people.collect{it.label + ': ' + it.currentMode}}
    unsubscribe()
    subscribe(people, "presence", presence)
}
```

122.4 Related documentation

- *Preferences and Settings* (page 285)
- *Events and Subscriptions* (page 325)
- *Working with Devices* (page 329)
- *Modes* (page 335)
- *Scheduling* (page 345)
- *Sending Notifications* (page 385)

122.5 Complete source code

The complete source code for this SmartApp can be found in the SmartThingsPublic GitHub repository [here](#).

Part XI

Web Services SmartApps

SmartApps may themselves be a web service, exposing a URL and any defined endpoints.

This allows external applications to make web API calls to a SmartApp, and get information about, or control, end devices.

Web Services SmartApps Overview

Integrating with SmartThings using SmartApps Web Services

In this guide, you will learn:

- The overall design of how Web Services SmartApps work.
 - Security measures taken to ensure access is only granted to trusted clients, and specific devices as chosen by the user.
 - The end user flow for external applications integrating with Web Services SmartApps.
-

123.1 Introduction

In designing a way to allow external systems API access, we wanted to give developers the flexibility they need, while ensuring that the customer understands why their account is being accessed through an external API, and has specifically authorized that access.

As such, we've designed an architecture and user experience around external API access that meets the following goals:

- It uses industry best practices such as OAuth2 to authenticate and authorize basic external API access.
 - It requires the end user (customer) to specifically authorize the access to specific devices.
 - It delivers a user experience that is easy to understand.
 - It delivers a developer experience that is easy to understand and implement.
-

123.2 Concepts

There are a couple of important concepts that need to be understood with respect to how SmartApps APIs work:

- All SmartApps APIs are authenticated using OAuth2.
 - When we talk about SmartApps APIs, we are referring to APIs that are exposed by SmartApps themselves.
 - SmartApps execute in a special security context, where they only have access to devices specifically authorized by the user at installation time. This is no different for SmartApps APIs.
-

123.3 How it works

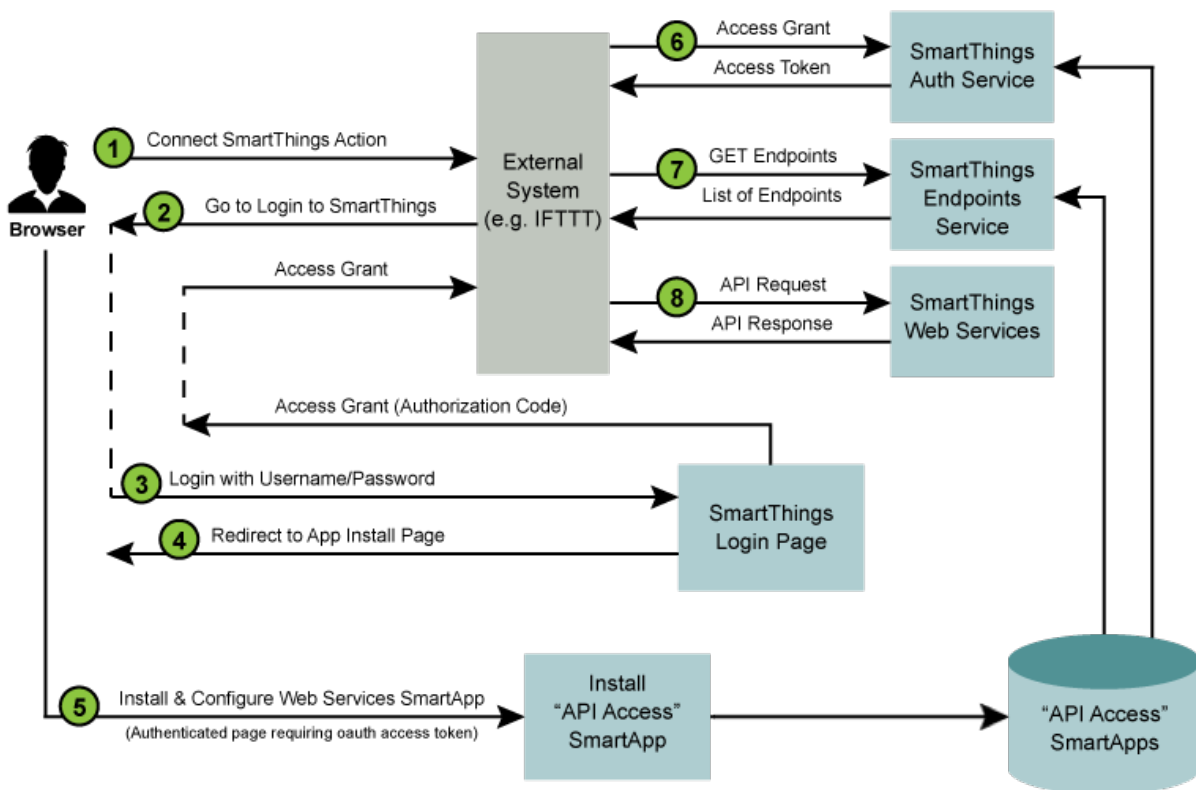
Our overall approach to API access requires the end user to authenticate and authorize the API access in two steps:

1. The installation of a SmartThings Web Services “SmartApp” into the user’s SmartThings Account/Location, along with specific device preferences that specify the devices to which the external system is being granted access.
2. The typical OAuth login flow grants the external system the OAuth access token.

It is important to understand that it is the SmartApp itself that exposes the API endpoints that are then used by the external system to integrate with SmartThings.

This approach is designed to ensure that an external system must have explicit access granted to the devices, before it can control those devices.

123.3.1 OAuth-integrated app installation flow



The diagram above outlines the following standard steps in the API Connection and Usage process:

1. A user of the external system takes some action that initiates a “Connect to SmartThings” flow. An example of this is an IFTTT user adding the SmartThings “channel”.
2. The external service will typically redirect to the SmartThings login page. The HTTP request to this page includes the required OAuth client ID (more details below), allowing our login page to recognize this as a login

request using OAuth.

3. The login page is displayed, and if the login is successful, a subsequent page is displayed that allows the now-authenticated user to install and configure the Web Services SmartApp that is associated with the client ID. When this step is complete, an authorization code is returned to the browser.
4. Typically, the authorization code is then given to the external system, and it is used (along with the OAuth client ID and client secret), to request an access token. The authorization code takes the place of the user credentials in this case, and is only valid for a single use. Once the external system has the OAuth access token, API requests can be made using this token.
5. The first API call that the external system should make is to the endpoints service. This service exists on a standard URL, and will return the specific URL that the external system should use (for this specific OAuth access token) to make all API requests.
6. Finally, the external system can use the specified endpoint URL and the provided OAuth2 access token to make API calls to the SmartApp providing the web services.

123.4 The end-user journey

Before discussing the specific steps to building a Web Services SmartApp, you should understand the end user experience.

123.4.1 Initiate connection from external system

The first step is to initiate the connection with the SmartThings cloud from the external web application. This is different for each web application, but is just a URL.

123.4.2 Authentication and authorization

The typical OAuth journey is the OAuth2 authorization code flow, initiated from the website of the external system, whereby the user is redirected to the SmartThings website. This is where they enter their SmartThings credentials, as shown below:

Already have SmartThings? Sign in here:

Email

Password:

[Forgot password?](#)

New to SmartThings? [Learn More](#) or [Get SmartThings](#) today.

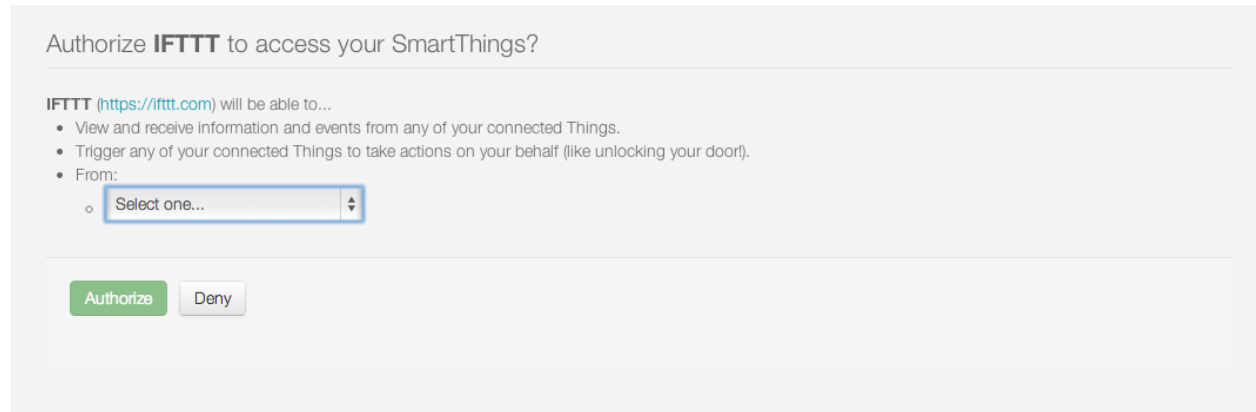
Copyright © 2016 Physical Graph Corporation. All rights reserved. | [Terms of Use](#) | [Privacy Policy](#) | [Benefits](#) | [Shop](#) | [Blog](#) | [Developers](#) | [Support](#)

Once authenticated with SmartThings, they will be prompted to specifically authorize access by the application.

123.4.3 Application configuration

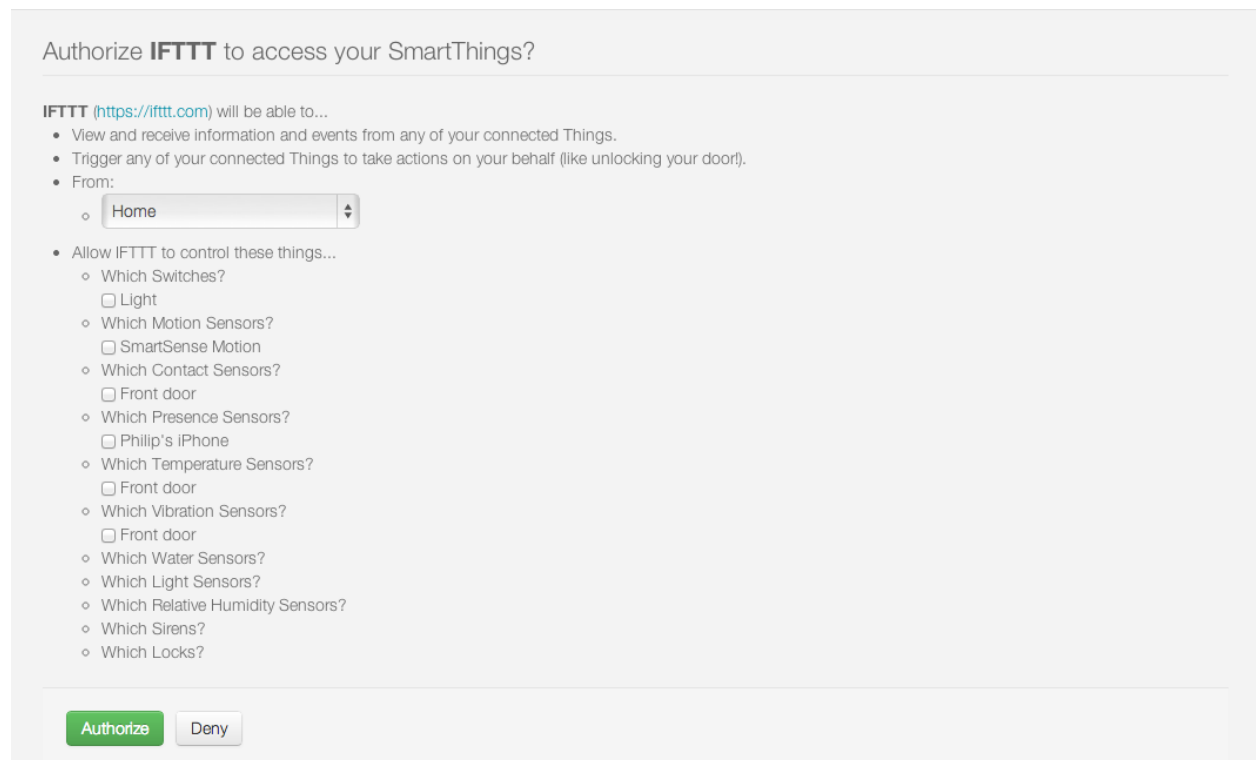
The user is prompted to configure the Web Services SmartApp that will be automatically installed. The user does not have to select the specific SmartApp, because it can be automatically identified by the OAuth client ID.

The first step in the application configuration process is to identify the Location in which the SmartApp will be installed.



The second step is to configure exactly which devices will be accessible through any external web services that are exposed by the SmartApp.

An example of the IFTTT SmartApp device selection options is shown below:



Finally, the user clicks on *Authorize* to complete both the authorization of the application and the installation of the SmartApp and the connection between the external system and the SmartThings Cloud is now complete.

Once the user authorizes access, the external system is provided with the OAuth authorization code, which is in turn used to request and receive an OAuth access token. Once the external system has the token, it can access the web services provided by the SmartApp.

Web Services Tutorial–SmartApp

This is the first part of two that will teach you how to build a Web Services SmartApp and a web application to illustrate the authorization flow.

124.1 Overview

In part 1 of this tutorial, you will learn:

- How to develop a Web Services SmartApp that exposes endpoints.
- How to call the Web Services SmartApp using simple API calls.

The source code for this tutorial is available [here](#).

Part 1 of this tutorial will build a simple SmartApp that exposes endpoints to get information about and control switches.

124.2 Create a new SmartApp

Create a new SmartApp in the IDE. Fill in the required fields, and make sure to click on *Enable OAuth in SmartApp* to receive an auto-generated client ID and secret.

Make sure to specify the redirect URI as this will be used to validate the authorization code request. For the purposes of this tutorial, simply type in `http://localhost:4567/oauth/callback`.

Note the Client ID and secret - they'll be used later (should you forget, you can get them by viewing the “App Settings” in the IDE).

124.3 Define preferences

SmartApps declare preferences metadata that is used at installation and configuration time, to allow the user to control what devices the SmartApp will have access to.

This is a configuration step, but also a security step, whereby the users must explicitly select what devices the SmartApp can control.

Web Services SmartApps are no different, and this is part of the power of this approach. The end user controls exactly what devices the SmartApp will have access to, and therefore what devices the external systems that consume those web services will have access to.

The preferences definition should look like this:

```
preferences {
  section ("Allow external service to control these things...") {
    input "switches", "capability.switch", multiple: true, required: true
  }
}
```

Also ensure that you have an `installed()` and `updated()` method defined (this should be created by default when creating a SmartApp). They can remain empty, since we are not subscribing to any device Events in this example.

You can learn more about Web Services SmartApp preferences [here](#) (page 434).

124.4 Specify endpoints

The `mappings` declaration allows developers to expose HTTP endpoints, and map the various supported HTTP operations to an associated handler.

Our SmartApp will expose two endpoints:

- The `/switches` endpoint will support a GET request. A GET request to this endpoint will return state information for the configured switches.
- The `/switches/:command` endpoint will support a PUT request. A PUT request to this endpoint will execute the specified command ("on" or "off") on the configured switches.

Here's the code for our mappings definition. This is defined at the top-level in our SmartApp (i.e., not in another method):

```
mappings {
  path("/switches") {
    action: [
      GET: "listSwitches"
    ]
  }
  path("/switches/:command") {
    action: [
      PUT: "updateSwitches"
    ]
  }
}
```

Note the use of variable parameters in our PUT endpoint. Use the `:` prefix to specify that the value will be variable. We'll see later how to get this value.

Go ahead and add empty methods for the various handlers. We'll fill these in in the next step:

```
def listSwitches() {}

def updateSwitches() {}
```

See the *Mapping endpoints* (page 435) documentation for more information.

124.5 GET switch information

Now that we've defined our endpoints, we need to handle the requests in the handler methods we stubbed in above.

Let's start with the handler for GET requests to the `/switches` endpoint. When a GET request to the `/switches` endpoint is called, we want to return the display name, and the current switch value (e.g., on or off) for the configured switch.

Our handler method returns a list of maps, which is then serialized by the SmartThings platform into JSON:

```
// returns a list like
// [[name: "kitchen lamp", value: "off"], [name: "bathroom", value: "on"]]
def listSwitches() {
    def resp = []
    switches.each {
        resp << [name: it.displayName, value: it.currentValue("switch")]
    }
    return resp
}
```

See the *Response handling* (page 438) documentation for more information on working with web request responses.

124.6 UPDATE the switches

We also need to handle a PUT request to the `/switches/:command` endpoint. `/switches/on` will turn the switches on, and `/switches/off` will turn the switches off.

If any of the configured switches does not support the specified command, we'll return a 501 HTTP error.

```
void updateSwitches() {
    // use the built-in request object to get the command parameter
    def command = params.command

    // all switches have the command
    // execute the command on all switches
    // (note we can do this on the array - the command will be invoked on every element
    switch(command) {
        case "on":
            switches.on()
            break
        case "off":
            switches.off()
            break
        default:
            httpError(400, "$command is not a valid command for all switches specified")
    }
}
```

Our example uses the endpoint itself to get the command. You can learn more about working with requests *here* (page 436).

124.7 Self-publish the SmartApp

Publish the app for yourself, by clicking on the *Publish* button and selecting *For Me*.

124.8 Run the SmartApp in the Simulator

Using the Simulator, we can quickly test our Web Services SmartApp.

Click the *Install* button in the Simulator, select a Location to install the SmartApp into, and select a switch.

Note that in the lower right of the Simulator there is an API token and an API endpoint URL:



Important: The base URL for of your SmartApp’s API endpoint will vary depending on the Location being installed into.

Be sure to copy the URL from the Simulator to ensure you have the correct URL!

We can use these to test making requests to our SmartApp.

124.9 Make API calls to the SmartApp

Using whatever tool you prefer for making web requests (this example will use curl, but [Apigee](#) is a good UI-based tool for making requests), we will call one of our SmartApp endpoints.

From the Simulator, grab the API endpoint. It will look something like this:

```
https://<BASE-URL>/api/SmartApp/Installation/1584595-3695-493E-AC01-80E93288-0000
```

Your installation will have a different, unique URL.

Important: The base URL for of your SmartApp’s API endpoint will vary depending on the Location being installed into.

Be sure to copy the URL from the Simulator to ensure you have the correct URL!

To get information about the switch, we will call the `/switch` endpoint using a GET request. You'll need to substitute your unique endpoint and API key.

```
curl -H "Authorization: Bearer <api token>" "<api endpoint>/switches"
```

This should return a JSON response like the following:

```
[{"name": "Kitchen 2", "value": "off"}, {"name": "Living room window", "value": "off"}]
```

To turn the switch on or off, call the `/switches` endpoint using a PUT request. Again, you'll need to substitute your unique endpoint and API key:

```
curl -H "Authorization: Bearer <api token>" -X PUT "<api endpoint>/switches/on"
```

Change the command value to `off` to turn the switch off. Try turning the switch on and off, and then using curl to get the status, to see that it changed.

124.10 Uninstall the SmartApp

Finally, uninstall the SmartApp using the *Uninstall* button in the IDE Simulator.

124.11 Summary

In this tutorial, you learned how to create a SmartApp that exposes endpoints to get information about, and control, a device. You also learned how to install the SmartApp in the Simulator, and then make API calls to the endpoint.

In the next part of this tutorial, we'll look at how an external application might interact with SmartThings using the OAuth2 flow (instead of simply using the Simulator and its generated access token).

Web Services SmartApp Tutorial–Authorization Flow

In Part 1 of this tutorial, you learned how to create a simple Web Services SmartApp, and install it in the IDE simulator, and make web requests to it.

In Part 2, we'll build a simple web application that will integrate with SmartThings and the WebServices SmartApp we created in Part 1.

125.1 Overview

In Part 2 of this tutorial, you will learn:

- How to get the API token.
- How to discover the endpoints of a Web Services SmartApp.
- How to make calls to the Web Services SmartApp.

The source code for this tutorial is available [here](#).

Important: Beginning February 1, 2016, only SmartApps approved and published by SmartThings can be installed via the OAuth flow discussed below.

For testing purposes, you will be able to install into your own account only.

For more information, see this [community post](#).

We will build a simple Sinatra application that will make calls to the Web Services SmartApp we built in Part 1.

If you're not familiar with Sinatra, you are encouraged to try it out. It's not strictly necessary, however, as our application will simply make web requests to get the API token and the endpoint.

Note: If Node is more your speed, check out the awesome SmartThings OAuth Node app written by community member John S (@schettj) [here](#). It shows how you can get an access token using the OAuth flow for a WebServices SmartApp using Node.

125.2 Prerequisites

Aside from completing Part 1 of this tutorial, you should have Ruby and Sinatra installed.

Visit the [Ruby](#) website to install Ruby, and the [Sinatra Getting Started Page](#) for information about installing Sinatra.

125.3 Bootstrap the Sinatra app

Create a new directory for the Sinatra app, and change directories to it:

```
mkdir web-app-tutorial
cd web-app-tutorial
```

In your favorite text editor*, create a new file called `server.rb` and paste the following into it, and save it.

**(If your favorite text editor is vim or emacs, then our hat's off to you. We're impressed - maybe even a bit intimidated. If your favorite editor is notepad, well... we're not as impressed, or intimidated. :@))*

```
require 'bundler/setup'
require 'sinatra'
require 'oauth2'
require 'json'
require "net/http"
require "uri"

# Our client ID and secret, used to get the access token
CLIENT_ID = ENV['ST_CLIENT_ID']
CLIENT_SECRET = ENV['ST_CLIENT_SECRET']

# We'll store the access token in the session
use Rack::Session::Pool, :cookie_only => false

# This is the URI that will be called with our access
# code after we authenticate with our SmartThings account
redirect_uri = 'http://localhost:4567/oauth/callback'

# This is the URI we will use to get the endpoints once we've received our token
endpoints_url = 'https://graph.api.smartthings.com/api/smartapps/endpoints'

options = {
  site: 'https://graph.api.smartthings.com',
  authorize_url: '/oauth/authorize',
  token_url: '/oauth/token'
}

# use the OAuth2 module to handle OAuth flow
client = OAuth2::Client.new(CLIENT_ID, CLIENT_SECRET, options)

# helper method to know if we have an access token
def authenticated?
  session[:access_token]
end

# handle requests to the application root
get '/' do
```

```
    %(<a href="/authorize">Connect with SmartThings</a>)
end

# handle requests to /authorize URL
get '/authorize' do
  'Not Implemented!'
end

# handle requests to /oauth/callback URL. We
# will tell SmartThings to call this URL with our
# authorization code once we've authenticated.
get '/oauth/callback' do
  'Not Implemented!'
end

# handle requests to the /getSwitch URL. This is where
# we will make requests to get information about the configured
# switch.
get '/getswitch' do
  'Not Implemented!'
end
```

Create your Gemfile - open a new file in your editor, paste the contents below in, and save it as Gemfile.

```
source 'https://rubygems.org'

gem 'sinatra'
gem 'oauth2'
gem 'json'
```

We'll use bundler to install our app. If you don't have it, you can learn how to get started [here](#).

Back at the command line, run bundle:

```
bundle install
```

You'll also want to set environment variables for your `ST_CLIENT_ID` and `ST_CLIENT_SECRET`.

Now, run the app on your local machine:

```
ruby server.rb
```

Visit <http://localhost:4567>. You should see a page with a link to "Connect with SmartThings".

We're using the [OAuth2 module](#) to handle the OAuth2 flow. We create a new `client` object, using the `client_id` and `client_secret`. We also configure it with the `options` data structure that defines the information about the SmartThings OAuth endpoint.

We've handled the root URL to simply display a link that points to the `/authorize` URL of our server. We'll fill that in next.

125.4 Get an authorization code

When the user clicks on the "Connect with SmartThings" link, we need to get our OAuth authorization code.

To do this, the user will need to authenticate with SmartThings, and authorize the devices this application can work with. Once that has been done, the user will be directed back to a specified `redirect_uri`, with the OAuth

authorization code. When we created the SmartApp in the first part of this tutorial, we set the redirect URI to `http://localhost:4567/oauth/callback`. It is important that the redirect URI in the SmartApp and the `redirect_uri` field in this Sinatra app match, as validation will occur with the authorization code request that will make sure these two URIs match. This will be used (along with the `client_id` and `client_secret`), to get the access token.

Important: When you self-publish a SmartApp, it is published and available in the Location that you published it. Since SmartThings is moving into the global space, the Location that you published your SmartApp corresponds to a specific server. This means your self-published SmartApp is only available on that server.

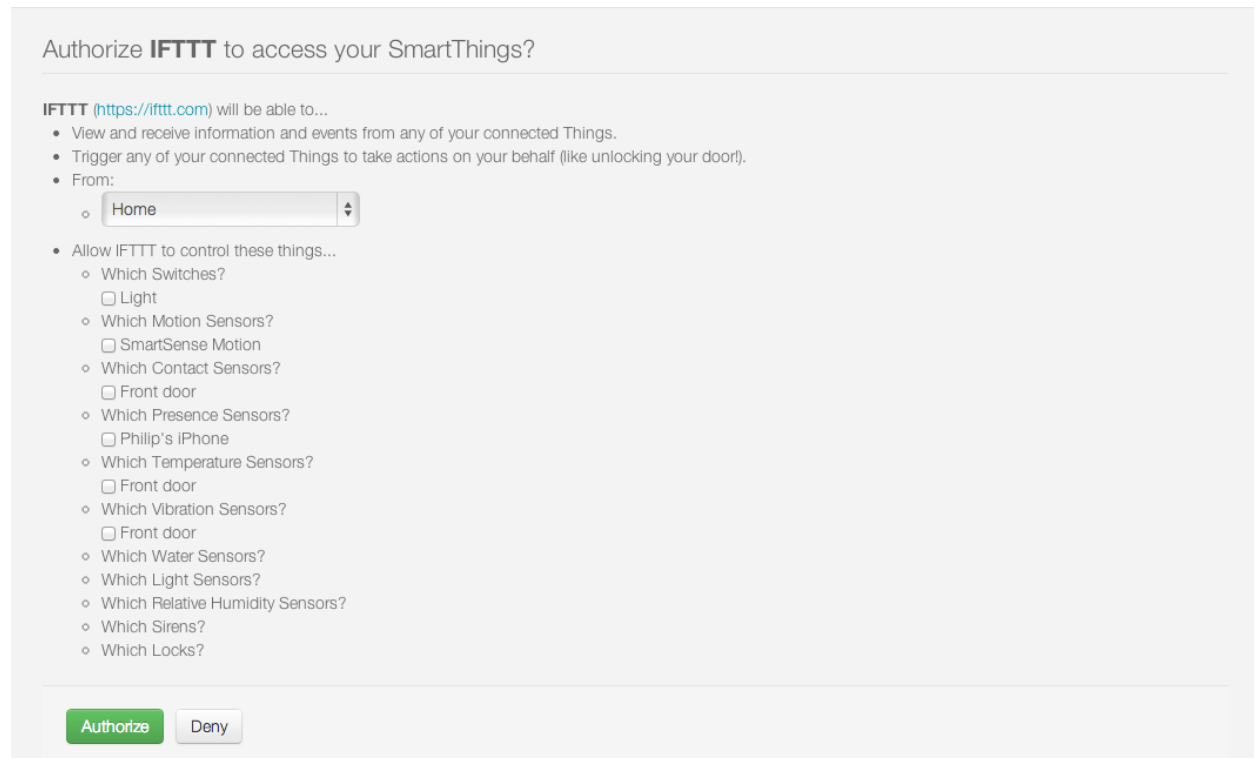
Replace the `/authorize` route with the following:

```
get '/authorize' do
  # Use the OAuth2 module to get the authorize URL.
  # After we authenticate with SmartThings, we will be redirected to the
  # redirect_uri, including our access code used to get the token
  url = client.auth_code.authorize_url(redirect_uri: redirect_uri, scope: 'app')
  redirect url
end
```

Kill the server if it's running (CTRL+C), and start it up again using `ruby server.rb`.

Visit `http://localhost:4567` again, and click the “Connect with SmartThings” link.

This should prompt you to authenticate with your SmartThings account (if you are not already logged in), and bring you to a page where you must authorize this application. It should look something like this:



The screenshot shows a web page titled "Authorize IFTTT to access your SmartThings?". Below the title, it says "IFTTT (https://ifttt.com) will be able to...". There are three bullet points: "View and receive information and events from any of your connected Things.", "Trigger any of your connected Things to take actions on your behalf (like unlocking your door).", and "From:". Under "From:", there is a dropdown menu with "Home" selected. Below that, there is a section "Allow IFTTT to control these things..." with a list of sensors and their corresponding checkboxes: "Which Switches?" (checkbox for Light), "Which Motion Sensors?" (checkbox for SmartSense Motion), "Which Contact Sensors?" (checkbox for Front door), "Which Presence Sensors?" (checkbox for Philip's iPhone), "Which Temperature Sensors?" (checkbox for Front door), "Which Vibration Sensors?" (checkbox for Front door), "Which Water Sensors?", "Which Light Sensors?", "Which Relative Humidity Sensors?", "Which Sirens?", and "Which Locks?". At the bottom of the page, there are two buttons: "Authorize" (green) and "Deny" (grey).

Click the *Authorize* button, and you will be redirected back your server.

You'll notice that we haven't implemented handling this URL yet, so we see “Not Implemented!”.

125.5 Get an access token

When SmartThings redirects back to our application after authorizing, it passes a `code` parameter on the URL. This is the code that we will use to get the API token we need to make requests to our Web Services SmartApp.

We'll store the access token in the session. Towards the top of `server.rb`, we configure our app to use the session, and add a helper method to know if the user has authenticated:

```
# We'll store the access token in the session
use Rack::Session::Pool, :cookie_only => false

def authenticated?
  session[:access_token]
end
```

Replace the `/oauth/callback` route with the following:

```
get '/oauth/callback' do
  # The callback is called with a "code" URL parameter
  # This is the code we can use to get our access token
  code = params[:code]

  # Use the code to get the token.
  response = client.auth_code.get_token(code, redirect_uri: redirect_uri, scope: 'app')

  # now that we have the access token, we will store it in the session
  session[:access_token] = response.token

  # debug - inspect the running console for the
  # expires in (seconds from now), and the expires at (in epoch time)
  puts 'TOKEN EXPIRES IN ' + response.expires_in.to_s
  puts 'TOKEN EXPIRES AT ' + response.expires_at.to_s
  redirect '/getswitch'
end
```

We first retrieve the access code from the parameters. We use this to get the token using the OAuth2 module, and store it in the session.

We then redirect to the `/getswitch` URL of our server. This is where we will retrieve the endpoint to call, and get the status of the configured switch.

Restart your server, and try it out. Once authorized, you should be redirected to the `/getswitch` URL. We'll start implementing that next.

125.6 Discover the endpoint

Now that we have the OAuth token, we can use it to discover the endpoint of our WebServices SmartApp.

Replace the `/getswitch` route with the following:

```
get '/getswitch' do
  # If we get to this URL without having gotten the access token
  # redirect back to root to go through authorization
  if !authenticated?
    redirect '/'
  end
end
```

```
token = session[:access_token]

# make a request to the SmartThings endpoint URI, using the token,
# to get our endpoints
url = URI.parse(endpoints_uri)
req = Net::HTTP::Get.new(url.request_uri)

# we set a HTTP header of "Authorization: Bearer <API Token>"
req['Authorization'] = 'Bearer ' + token

http = Net::HTTP.new(url.host, url.port)
http.use_ssl = (url.scheme == "https")

response = http.request(req)
json = JSON.parse(response.body)

# debug statement
puts json

# get the endpoint from the JSON:
uri = json[0]['uri']

'<h3>JSON Response</h3><br/>' + JSON.pretty_generate(json) + '<h3>Endpoint</h3><br/>' + uri
end
```

The above code simply makes a GET request to the SmartThings API endpoints service at `https://graph.api.smartthings.com/api/smartapps/endpoints`, setting the "Authorization" HTTP header with the API token.

The response is JSON that contains (among other things), the endpoint of our SmartApp. The JSON that is returned contains a key called `uri` that we will use to build our endpoint URLs. There are other URL keys in the JSON, but the `uri` key is specific to the server that your SmartApp is on. Always use the `uri` key or `base_uri` for your endpoints. For this step, we just display the JSON response and endpoint in the page.

By now, you know the drill. Restart your server, refresh the page, and click the link (you'll have to reauthorize). You should then see the JSON response and endpoint displayed on your page.

125.7 Make API calls

Now that we have our token and endpoint, we can make API calls to our SmartApp.

As you may have guessed by the URL path, we're just going to display the name of the switch, and it's current status (on or off).

Remove the line at the end of the `getswitch` route handler that outputs the response HTML, and add the following:

```
# now we can build a URL to our WebServices SmartApp
# we will make a GET request to get information about the switch
switchUrl = uri + '/switches'

# debug
puts "SWITCH ENDPOINT: " + switchUrl

getSwitchURL = URI.parse(switchUrl)
getSwitchReq = Net::HTTP::Get.new(getSwitchURL.request_uri)
```

```
getSwitchReq['Authorization'] = 'Bearer ' + token
getSwitchHttp = Net::HTTP.new(getSwitchURI.host, getSwitchURI.port)
getSwitchHttp.use_ssl = true

switchStatus = getSwitchHttp.request(getSwitchReq)

'<h3>Response Code</h3>' + switchStatus.code + '<br/><h3>Response Headers</h3>' + switchStatus.headers
```

The above code uses the endpoint (obtained from the *uri* key in our JSON response above) for our SmartApp to build a URL, and then makes a GET request to the `/switches` endpoint. It simply displays the the status, headers, and response body returned by our WebServices SmartApp.

Restart your server and try it out. You should see status of your configured switches displayed!

125.8 Summary

In the second part of this tutorial, we learned how an external application can work with SmartThings by getting an access token, discover endpoints, and make API calls to a WebServices SmartApp.

You are encouraged to explore further with this sample, including making different API calls to turn the configured switch on or off.

The SmartApp

A Web Services SmartApp exposes endpoints that third parties can make REST calls to. It can then do anything a normal SmartApp can do - get device status, actuate devices, etc. - and send a response back to the calling client.

126.1 Enable OAuth

For a SmartApp to expose endpoints that can receive REST calls from a third party, OAuth must be enabled.

OAuth can be enabled for a SmartApp via the *App Settings* page. In the *OAuth* section, check the box to enable OAuth. A client ID and secret will be generated for this SmartApp. These will be used as part of the OAuth flow to obtain an access token for this SmartApp.

There is an option to specify a Redirect URI. This URI will be used to validate the `redirect_uri` passed in with the request for the authorization code. The value of this field can be a single value, or a comma-delimited list of values. For example:

```
http://myserverhostname.com
```

or

```
http://myserverhostname1.com,http://myserverhostname2.com,http://myserverhostname3.com
```

During validation, the `redirect_uri` passed in with the authorization code request will be checked against the URIs defined in this field. The port does matter during validation. If there is no match, validation will fail with the following error:

```
OAuth2 Error
```

```
error="invalid_grant", error_description="Invalid redirect: http://myserverhostname.com/oauth/callback"
```

You can also set the *Client Display Name* and *Client Display Link*. These will be used on the SmartThings Authorization page to inform the user who is requesting access to their devices.

OAuth

Client ID:
Public client ID for accessing this SmartApp via its REST API
Generate New Client ID

Client Secret:
Confidential secret key for accessing this SmartApp via its REST API
Generate New Client Secret

Redirect URI: (optional)
URI of authorized server used for redirect URL validation. If a provided redirect URL doesn't match this URI the authorization request will be rejected.

Display Name: (optional)
Company or product name representing this application that is displayed to the user during the authorization process

Display Link: (optional)
URL of the website representing this application, provided to the user during the authorization

126.2 Preferences

Part of the *Authorization Flow* (page 441) that installs the SmartApp requires the user to authorize specific devices that the third party can interact with. The types of devices that may be authorized for Web Services SmartApps are controlled through the SmartApp's preferences.

The intent of the Authorization page is to simply allow the user to authorize specific devices. This can be accomplished in one of two ways:

- Specify a simple, single page that allows the user to select from a set of devices, or
- Specify a specific preferences page to be used by the Authorization web page.

An example of a simple, single page that will allow the user to select from a set of devices:

```
preferences {
    section("Control these switches...") {
        input "switches", "capability.switch"
    }
    section("Control these motion sensors...") {
        input "motion", "capability.motionSensor"
    }
}
```

Here is an example that specifies a specific page to be used during authorization, using the `oauthPage` option to preferences:

Warning: Currently, using required inputs does not work inside of page declarations when using `oauthPage`. This is a known issue and is currently being worked on. We recommend using non-required inputs, by explicitly setting `required: false`, when using `oauthPage` pages.

```

preferences(oauthPage: "deviceAuthorization") {
  // deviceAuthorization page is simply the devices to authorize
  page(name: "deviceAuthorization", title: "", nextPage: "instructionPage",
    install: false, uninstall: true) {
    section("Select Devices to Authorize") {
      input "switches", "capability.switch", title: "Switches:", required: false
      input "motions", "capability.motionSensor", title: "Motion Sensors:", required: false
    }
  }

  page(name: "instructionPage", title: "Device Discovery", install: true) {
    section() {
      paragraph "Some other information"
    }
  }
}

```

If you require additional, non-device preferences inputs, you can use dynamic pages. The `oauthPage` must be a static (non-dynamic) page, and be the first page displayed:

```

preferences(oauthPage: "deviceAuthorization") {
  // deviceAuthorization page is simply the devices to authorize
  page(name: "deviceAuthorization", title: "", nextPage: "otherPage",
    install: false, uninstall: true) {
    section("Select Devices to Authorize") {
      input "switches", "capability.switch", title: "Switches:", required: false
      input "motions", "capability.motionSensor", title: "Motion Sensors:", required: false
    }
  }

  page(name: "otherPage")
}

def otherPage() {
  dynamicPage(name: "otherPage", title: "Other Page", install: true) {
    section("Other Inputs") {
      input "sometext", "text"
      input "sometime", "time"
    }
  }
}

```

126.3 Mapping endpoints

To expose a callable endpoint in your SmartApp, use mappings. Specify the various endpoints using `path`, and specify the supported HTTP methods (GET, PUT, POST, and DELETE). Each action specified is associated with the name of a method that will handle the request.

```

mappings {
  path("/foo") {
    action: [
      GET: "getFoo",

```

```
        PUT: "putFoo",
        POST: "postFoo",
        DELETE: "deleteFoo"
    ]
}
path("/bar") {
    action: [
        GET: "getBar"
    ]
}
}

def getFoo() {}
def putFoo() {}
def postFoo() {}
def deleteFoo() {}
def getBar() {}
```

There is no limit to the number of endpoints a SmartApp exposes, but the path level is restricted to four levels deep (i.e., /level1/level2/level3/level4).

You can specify variable URL path parameters using the `:` prefix in the path:

```
mappings {
    path("/foo/:param1/:param2") {
        action: [GET: "getFoo"]
    }
}
```

126.4 Request handling

When a request is made to one of the SmartApp's endpoints, its associated request handler method will be called.

Every request handler method has available to it a `request` object that represents information about the request, and a `params` object that contains information about the request parameters.

Important: All request or path parameters should be validated in your request handler. **Never** allow parameters to arbitrarily execute device commands or otherwise modify data.

126.4.1 Path variables

Any path variables you defined in the path are available via the injected `params` object:

```
mappings {
    path("/switches/:command") {
        action: [PUT: "updateSwitches"]
    }
}

def updateSwitches() {
    def cmd = params.command
    log.debug "command: $cmd"
```



```

switch(cmd) {
    case "on":
        // handle on command
        break
    case "off":
        // handle off command
        break
    default:
        httpError(501, "$command is not a valid command for all switches specified")
}

```

126.4.2 Query parameters

URL query parameters sent on the request are available via the `params` object:

```

def someHandler() {
    // this endpoint can accept the "foo" query parameter
    def fooParam = params.foo
    log.debug "foo parameter: $foo"
}

```

126.4.3 Request body parameters

SmartThings supports JSON or XML request body parameters. They can be accessed via `request.JSON` and `request.XML`:

```

// json on request: '{"foo": "bar"}'
def someJSONHandler() {
    def fooJSON = request.JSON?.foo
    log.debug "foo json: $fooJSON"
}

// xml on request: '<foo>bar</foo>'
def someXMLHandler() {
    def fooXML = request.XML?.foo
    log.debug "foo xml: $fooXML"
}

```

Tip: Use the `?` (safe navigation operator) to avoid a `NullPointerException` if the request JSON or XML is null (in case the request did not send JSON or XML).

The JSON available on the `request` will be the result of calling `new JsonSlurper().parseText()`. You can learn more about working with JSON in Groovy [here](#).

Similarly, the XML on `request` is the result of calling `new XmlSlurper().parseText()`. Learn more about working with XML in Groovy [here](#).

126.5 Response handling

126.5.1 Defaults

Each HTTP method (GET, PUT, POST, DELETE) request handler returns a default response. Some request handlers may return a map that will be serialized to JSON on the response, and some may specify their own response by using the `render()` method:

Request Method	Default HTTP Response Code	JSON Serialization Support	<code>render()</code> support
GET	200 OK	yes	yes
POST	201 Created	yes	yes
PUT	204 No Content	no	no
DELETE	204 No Content	no	no

126.5.2 Automatic JSON serialization

GET and POST request handlers may return a map, which will be serialized to JSON and returned to the client with `Content-Type: application/json`:

```

mappings {
    path("/test") {
        action: [
            GET: "responseTest",
            POST: "responseTest"
        ]
    }
}

def responseTest() {
    // a map is serialized to JSON and returned on the response
    return [data: "test"]
}

```

The response of executing a GET or POST request on the `/test` endpoint results in the following:

```

HTTP/1.1 200 OK
Content-Type: application/json charset=utf-8
Date: Tue, 29 Mar 2016 13:53:14 GMT
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=XXXXXXXXXXXXXXXX-n1; Path=/; Secure HttpOnly
X-RateLimit-Current: 0
X-RateLimit-Limit: 250
X-RateLimit-TTL: 60
transfer-encoding: chunked
Connection: keep-alive

{"data": "test"}

```

126.5.3 Using `render()` to control the response

GET and POST request handlers also support the ability to return a custom response using the `render()` (page 927) method:

```

mappings {
  path("/test") {
    action: [
      GET: "responseTest",
      POST: "responseTest"
    ]
  }
}

def responseTest() {
  def html = """
  <!DOCTYPE html>
  <html>
    <head><title>Some Title</title></head>
    <body><p>Testing</p></body>
  </html>"""

  render contentType: "text/html", data: html, status: 200
}

```

The response of executing a GET or POST request on the `/test` endpoint results in the following:

```

HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Date: Tue, 29 Mar 2016 15:00:32 GMT
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=1A4382D4BDFCCB31CD6C4EF3C2E3D693-n5; Path=/; Secure; HttpOnly
Vary: Accept-Encoding
X-RateLimit-Current: 0
X-RateLimit-Limit: 250
X-RateLimit-TTL: 60
transfer-encoding: chunked
Connection: keep-alive

<!DOCTYPE html>
<html>
  <head><title>Some Title</title></head>
  <body><p>Testing</p></body>
</html>

```

If not specified, the `contentType` will be “application/json”, and the `status` will be 200.

126.6 Error handling

126.6.1 Default errors

The following errors may be returned by the SmartThings platform:

HTTP Response Code	Error Message	Cause
401 (Unauthorized)	{“error”: “invalid_token”, “error_description”: “<TOKEN>”}	Invalid token for the SmartApp installation.
403 (Forbidden)	{“error”:true, “type”:”AccessDenied”, “message”:”This request is not authorized by the specified access token”}	No installed SmartApp can be found associated with the token.
404 (Not Found)	{“error”:true,”type”:”SmartAppException”, “message”:”Not Found”}	The endpoint path requested does not exist.
405 (Method Not Allowed)	{“error”:true,”type”:”SmartAppException”, “message”:”Method Not Allowed”}	An endpoint path was called but no request handler is defined for the specified request method (e.g., issuing a POST request to an endpoint path that only handles GET requests)
429 (Too Many Requests)	{“error”: true, “type”: “RateLimit”, “message”: “Please try again later”}	The rate limit for this SmartApp installation has been exceeded. See the Web services rate limit headers (page 613) documentation for more information.
500 (Server Error)	{“error”:true, “type”:”<EXCEPTION-TYPE>”, “message”: “An unexpected error has occurred”}	An unhandled exception occurred in the processing of the request. Check the SmartThings live logging to debug.

126.6.2 Custom errors

If your endpoint needs to send an error response, use the `httpError()` (page 921) method:

```
def someHandler() {
    def foo = request.JSON?.foo

    if (!foo) {
        httpError(400, "Foo parameter required")
    }
}
```

A `SmartAppException` will be thrown, and a response will be sent to the client with the specified HTTP code. The body of the response will be `application/json`, and look like this:

```
{
  "error":true,
  "type":"SmartAppException",
  "message":"your error message"
}
```

You should send appropriate error codes and messages for any errors.

Authorization

To make REST requests to a SmartApp, the client must establish a trusted relationship using an implementation of the OAuth2 Authorization Code Flow.

127.1 Overview

The general flow is:

1. Request an authorization code.
2. Use the code to request an access token.
3. Get the endpoint URI for the SmartApp.
4. Make REST calls to the SmartApp using the endpoint URI.

As part of the authorization flow, the SmartApp will be installed to the user's selected Location.

Important: Beginning February 1, 2016, only SmartApps approved and published by SmartThings can be installed via the OAuth flow discussed below.

For testing purposes, you will be able to install into your own account only.

For more information, see this [community post](#).

Note: Regardless of the server the SmartApp is actually published to, `https://graph.api.smarthings.com` should be used to obtain the authorization code, access token, and endpoints.

127.2 Get authorization code

Authorization URL: `https://graph.api.smarthings.com/oauth/authorize`

To obtain an authorization code, make a GET request to `https://graph.api.smarthings.com/oauth/authorize:`

```
GET https://graph.api.smartthings.com/oauth/authorize?
    response_type=code
    client_id=YOUR-SMARTAPP-CLIENT-ID
    scope=app
    redirect_uri=YOUR-SERVER-URI
```

The following parameters are required:

parameter	value
response_type	Use code to obtain the authorization code.
client_id	The OAuth client ID of the SmartApp.
scope	This should always be “app” for this authorization flow.
redirect_uri	The URI of your server that will receive the authorization code. This URI must match one of the redirect URIs specified in the SmartApp settings, otherwise validation will fail.

This will require the user to log in with their SmartThings account credentials, choose a Location, and select what devices may be accessed by the third party.

The authorization code expires 24 hours after issue.

127.3 Get access token

Token URL: `https://graph.api.smartthings.com/oauth/token`

Use the code you received to obtain the access token:

```
POST https://graph.api.smartthings.com/oauth/token HTTP/1.1
Host: graph.api.smartthings.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=YOUR_CODE&client_id=YOUR_CLIENT_ID&client_secret=YOUR_CLIENT_SECRET
```

The content-type header of your request should be of the type `application/x-www-form-urlencoded`. The following form parameters are required:

parameter	value
grant_type	This is always “authorization_code” for this flow.
code	The code you received.
client_id	The client ID for the SmartApp.
client_secret	The client secret for the SmartApp.
redirect_uri	The URI of the server that will receive the token. This must match the URI you used to obtain the authorization code.

A successful response will look like this:

```
{
  "access_token": "XXXXXXXXXXXX",
  "expires_in": 157679999,
  "token_type": "bearer"
}
```

The `expires_in` response is the time, in seconds from now, that this token will expire.

Once you have the token, it must be stored securely in the application.

127.4 Get SmartApp endpoints

You can use the token to request the callable endpoints of the SmartApp, by making a GET request to `https://graph.api.smarththings.com/api/smartapps/endpoints`. The access token should be supplied via a `Authorization: Bearer` header:

```
GET -H "Authorization: Bearer ACCESS-TOKEN" "https://graph.api.smarththings.com/api/smartapps/endpoints"
```

A successful response will return a list of all installed SmartApps for the `clientID` associated with the given access token.

```
{
  "oauthClient": {
    "clientId": "CLIENT-ID"
  },
  "location": {
    "id": ID,
    "name": "LOCATION-NAME"
  },
  "uri": "BASE-URL/api/smartapps/installations/INSTALLATION-ID",
  "base_url": "BASE-URL",
  "url": "/api/smartapps/installations/INSTALLATION-ID"
}
```

Important: The `base_url` (and base URL of the `uri`) will vary depending upon the server the SmartApp is being installed to.

SmartApps may be installed into any number of servers depending upon the location of the end-user. You should always use the `uri` and `base_url` to find the location this SmartApp can be reached at.

Do not assume that the SmartApp will be installed on `https://graph.api.smarththings.com`.

127.5 Make REST calls

Using the `uri` returned from `/api/smartapps/endpoints`, you can then make REST calls the SmartApp.

Simply append any paths your SmartApp declares in its `mappings` to make the appropriate request.

For example, assuming a `mappings` definition like this:

```
mappings {
  path("/switches") {
    action: [GET: "getSwitches"]
  }
}
```

```
}  
  
def getSwitches() {  
    // ...  
}
```

And a URI of `https://graph.api.smarthings.com/api/smartapps/installations/12345`, you can make a request to the `/switches` endpoint like this:

```
curl -H "Authorization: Bearer ACCESS-TOKEN" -X GET "https://graph.api.smarthings.com/api/smartapps,
```

Troubleshooting

128.1 General

Developing and testing Web Services SmartApps can be tricky, in large part due to the nature of the OAuth process. Here are some general tips and strategies to help you be successful:

- Make you sure you have read and understand the *Web Services Authorization* (page 441) documentation.
- Remember that only SmartApps published by SmartThings can be installed into general user accounts. If you self-published the SmartApp, *only the account who published it can install the SmartApp for testing purposes*.
- Trying to complete the OAuth process through the browser, without exposing a callable URL to receive the token, will not work. Read through the *Web Services SmartApp Tutorial–Authorization Flow* (page 425) to see how this can be done.
- Understand that to make API calls to the SmartApp, you must first *make a REST call to obtain the specific URL for the installed SmartApp* (page 443). This should always be made to `https://graph.api.smarthings.com/api/smartapps/endpoints`, *regardless of the specific server the SmartApp is installed into*.

128.2 Errors during installation

When choosing a Location and selecting devices to authorize, there are some common errors that may occur.

128.2.1 “<clientID> is not associated with a SmartApp in Location” after selecting Location

Problem When attempting to install a Web Services SmartApp via the OAuth flow, SmartThings looks for a SmartApp published to the specific server for that Location with that Client ID. This error results from either the SmartApp not being published to the server that the user is installing into, or from trying to install a Web Services SmartApp into an account that did not publish the SmartApp.

Solution If the SmartApp was self-published, make sure you are using the same account to install into (only Web Service SmartApps published by SmartThings may be installed into other user accounts). If it is the same account, and you are trying to install into a different Location, ensure the SmartApp is published on that Location as well (this will require handling different OAuth Client ID and Secret).

If this is a SmartApp published by SmartThings, contact support@smarthings.com.

128.2.2 “Please select at least one device to authorize” error after clicking Authorize

Problem If you have selected devices to authorize, this error likely indicates that an exception occurred during the installation process itself (in the `installed()` or `updated()` methods).

Solution Check Live Logging for any exceptions, and look at any code executing in the `installed()` or `updated()` methods for possible bugs.

Part XII

Device Handlers

Device Handlers are the virtual representation of a physical device.

If you are new to writing Device Handlers, start with the [Quick Start](#) (page 451).

After that, read the [Overview](#) (page 459) for a broad discussion about Device Handlers and where they fit in the SmartThings architecture.

The rest of the guide discusses the various components of Device Handlers primarily targeted for Hub-connected (ZigBee or Z-Wave) devices (though the common Device Handler principles and patterns apply to other devices as well).

Note: This guide discusses Hub-connected Device Handlers. For information about LAN- and Cloud-connected Device Handlers, see [this guide](#)

Table of Contents:

Quick Start

A Device Handler is a representation of a physical device in the SmartThings platform. It is responsible for communicating between the actual device and the SmartThings platform.

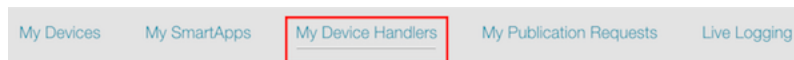
Alternately, a Device Handler can also be associated with a Virtual Device when a physical device is not yet available. This section will walk you through creating your first custom Device Handler and testing it with a Virtual Device.

Warning: Before you proceed, ensure that you are on the correct Location on IDE. Follow the prerequisites described in *Prerequisites* (page 179).

If you are new to SmartThings development, consider starting with the *Getting Started* (page 93) material.

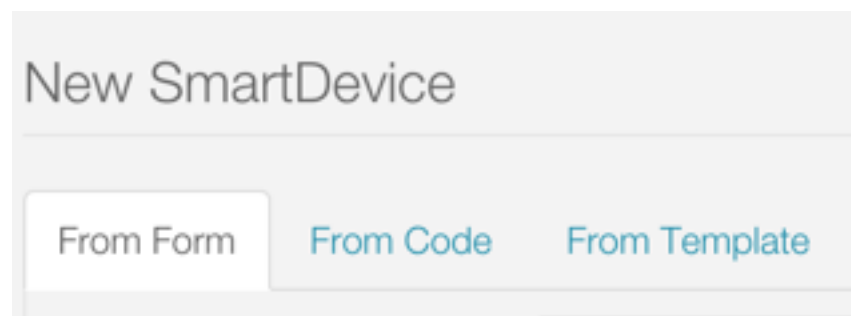
129.1 Create a new Device Handler

From IDE click on the *My Device Handlers* link on the top menu. Here you will see all your Device Handlers, if you have any.



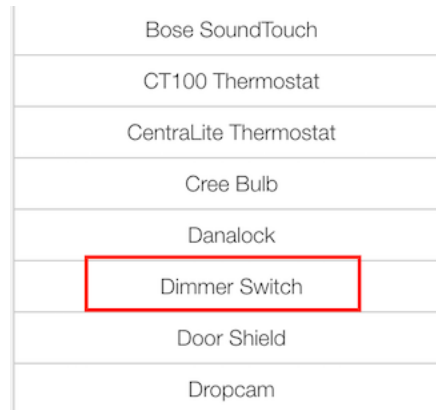
Create a new Device Handler by clicking on the *+Create New Device Handler* button in the upper-right of the page.

You will see a form for creating a new Device Handler. Note the tabs at the top of the form, showing different options for creating a new Device Handler:



Select the *From Template* tab.

We are going to create a new Device Handler from the Dimmer Switch template. Click on the *Dimmer Switch* in the menu on the left.



You will now see the Dimmer Switch Device Handler code on the right.

Take a minute to look at the code and its structure. Don't worry about the details yet - for now, just take note of the anatomy of the Device Handler:

```
Dimmer Switch
by SmartThings

* on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See
the License
* for the specific language governing permissions and limitations under the License.
*
*/
metadata {
    definition (name: "Dimmer Switch", namespace: "smarththings", author: "SmartThings") {
        capability "Switch Level"
        capability "Actuator"
        capability "Indicator"
        capability "Switch"
        capability "Polling"
        capability "Refresh"
        capability "Sensor"
        capability "Health Check"

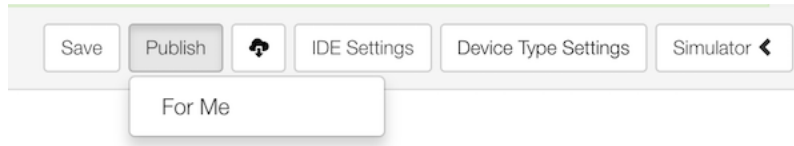
        fingerprint mfr:"0063", prod:"4457", deviceJoinName: "GE In-Wall Smart Dimmer "
        fingerprint mfr:"0063", prod:"4944", deviceJoinName: "GE In-Wall Smart Dimmer "
        fingerprint mfr:"0063", prod:"5044", deviceJoinName: "GE Plug-In Smart Dimmer "
        fingerprint mfr:"0063", prod:"4944", model:"3034", deviceJoinName: "GE In-Wall Smart

Fan Control"
    }

    simulator {
```

Next, make a few changes to this code to make it yours. In the definition method, change the `name` from “Dimmer Switch” to something like “My Dimmer Switch”, the `namespace` to your github user account (or you can leave it blank), and the `author` to your name.

Click the *Create* button below the editor, and then click *Publish* and *For Me* on the next screen.



129.2 Create a Virtual Device

Next, we will create a Virtual Device and associate it with the Device Handler we just created above.

From the top menu of the IDE, click on the *My Devices*.



Click on *+New Device* on the top-right. This will take you to *Create Device* page.

Follow below steps to fill the above *Create Device* form:

Name Your Virtual Device, preferably something that's indicative of the type of the device, such as "Virtual Dimmer Switch".

Label Optional, but you can have something like "virtual-dimmer-switch".

Zigbee Id Can be blank.

Device Network Id Should be a unique ID that identifies your Virtual Device. Make sure this ID doesn't conflict with any other device Ids. Put in "VIRTDIMMERS01".

Type Pulldown menu lists available Device Handlers. Note that all your custom Device Handlers are listed at the bottom of the pulldown list. Scroll down the list and select the customer Device Handler that you created above.

Version Option should be *Published*.

Location Must be your Hub Location.

Hub Your Hub name associated with the above Location.

Group Not selectable.

Click *Create*.

You will see *virtual-dimmer-switch* device appear instantly in your SmartThings mobile app, in the *Things* screen of the “My Home” view.

129.3 Test your Device Handler with Virtual Device

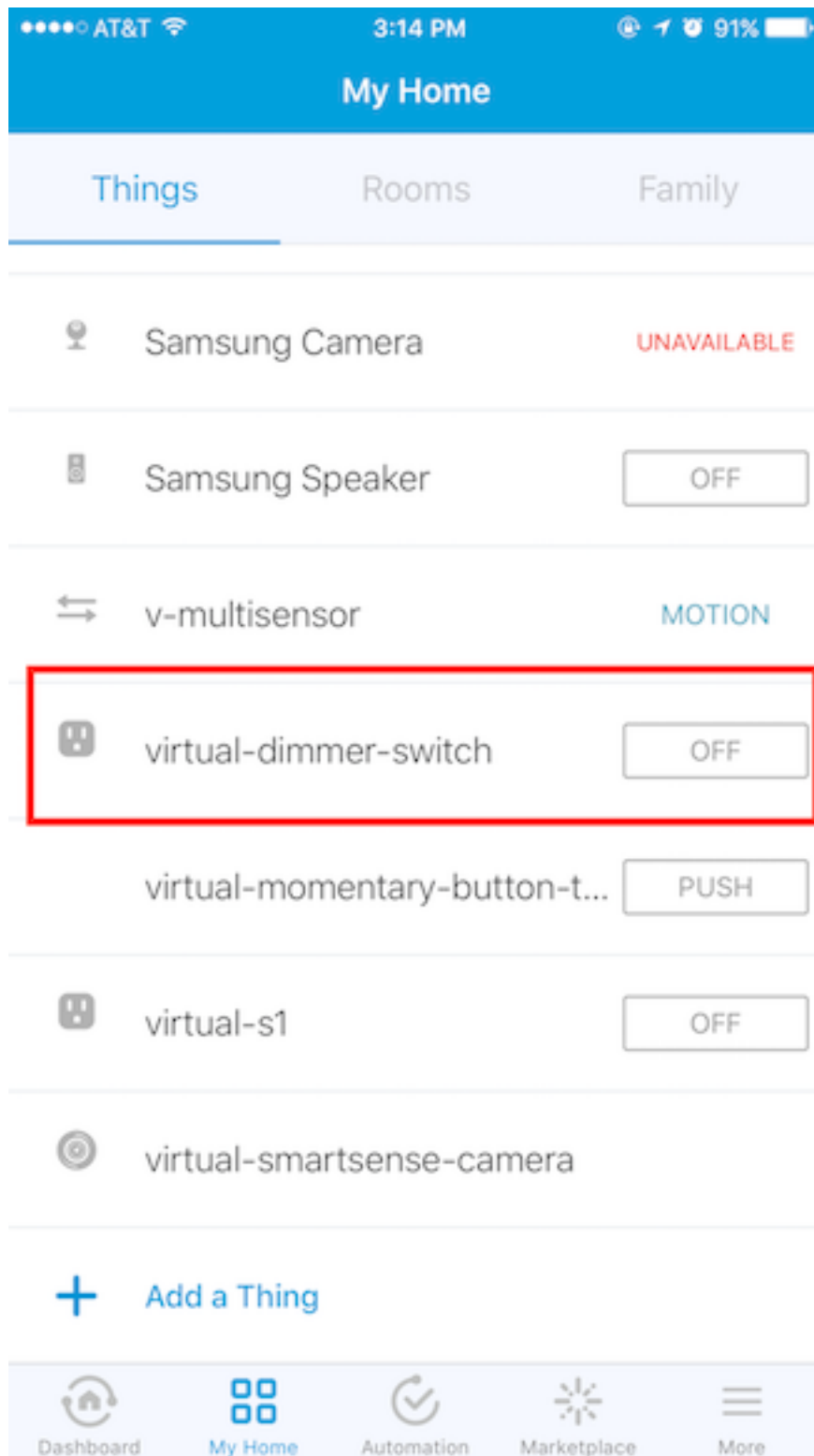
With the Virtual Dimmer you just created you can test your Device Handler. From your SmartThings mobile app, tap on the *OFF* tile of **virtual-dimmer-switch** to turn it *ON*.

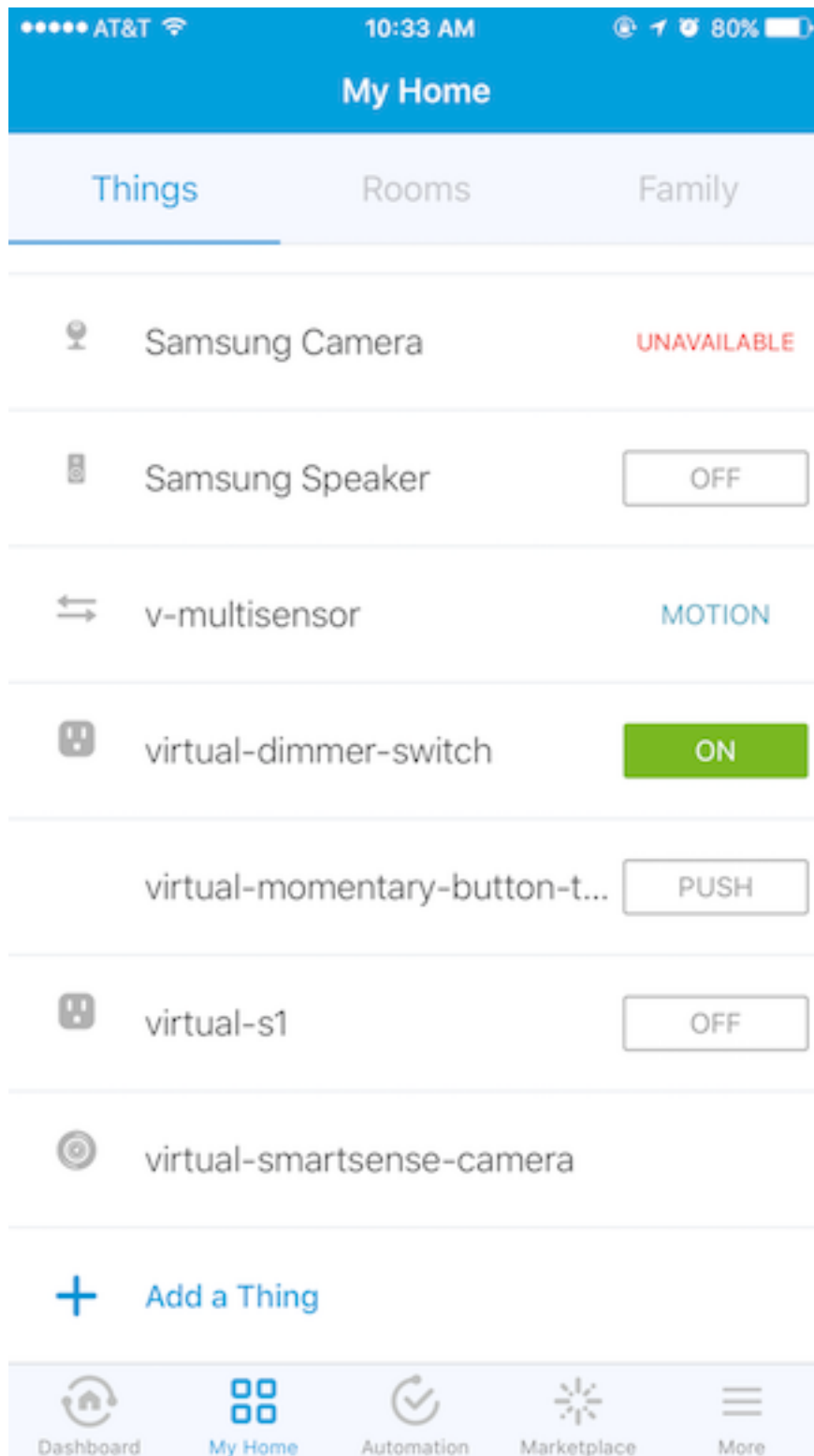
Next, tap on the **virtual-dimmer-switch** to open the detail view and test the tiles.

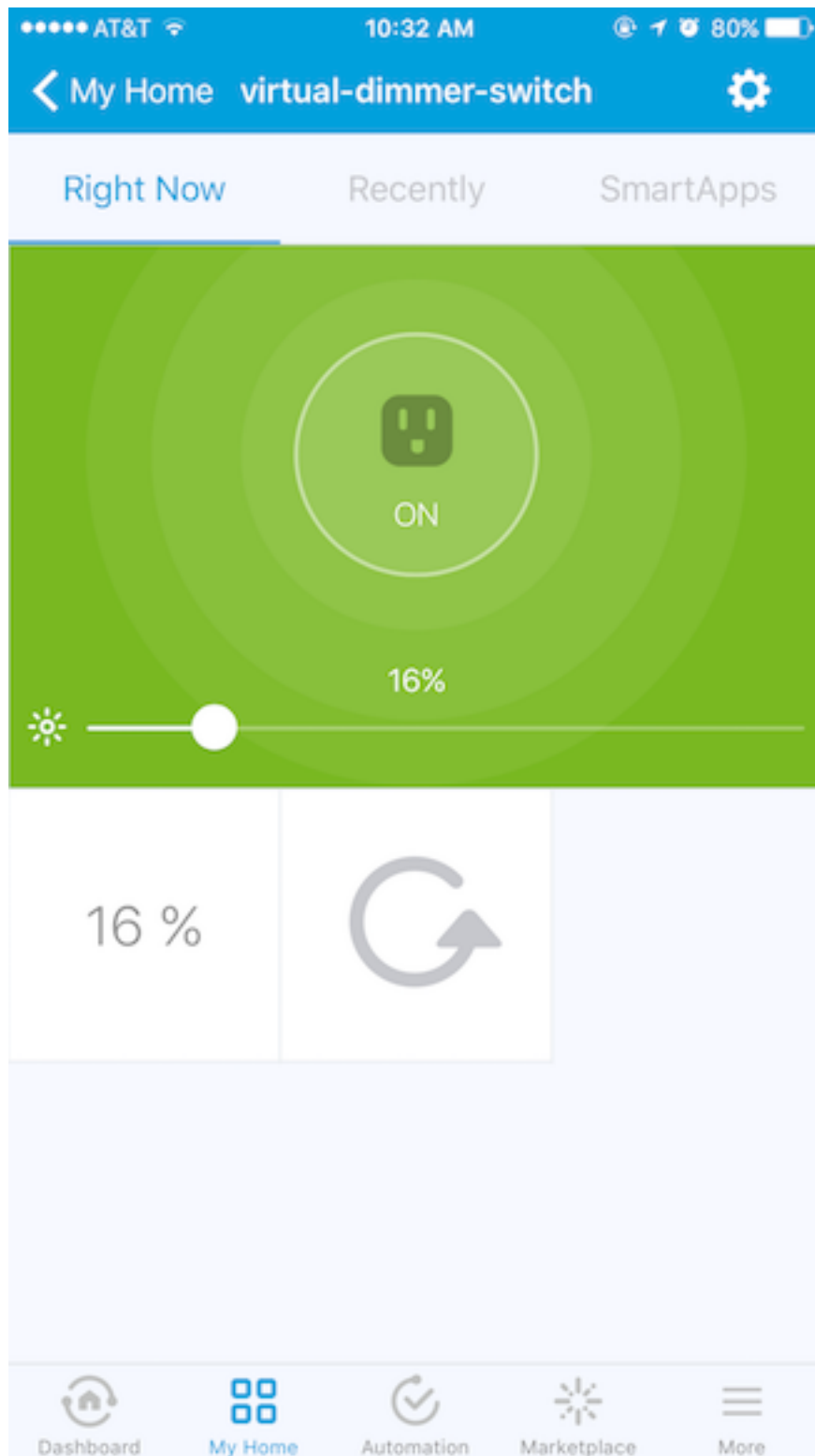
Note: While the Simulator is useful and necessary for testing how the Device Handler handles incoming messages, we recommended that you test on the mobile app with Virtual Devices wherever possible.

129.4 Next steps

Now that you have created and installed your first Device Handler with a Virtual Device, use the rest of this guide to learn more.







Overview

The SmartThings architecture provides a unique abstraction of devices from their distinct capabilities and attributes in a way that allows developers to build applications that are insulated from the specifics of which device they are using. For example, there are lots of wirelessly controllable “switches”. A switch is any device that can be turned On or Off.

When a SmartApp interacts with the virtual representation of a device, it knows that the device supports certain actions based on its capabilities. A device that has the “switch” capability must support both the “on” and “off” actions. In this way, all switches are the same, and it doesn’t matter to the SmartApp what kind of switch is actually involved.

This virtual representation of the device is called a Device Handler.

Note: This layer of abstraction is key to the successful function and flexibility of the SmartThings platform. Architecturally, device handlers are the bridge between generic capabilities and the device or protocol specific interface actually used to communicate with the device.

The diagram below depicts where device handlers sit in the SmartThings architecture.

In the example shown above, the job of the Device Handler (that is implementing the “switch” capability) is to parse incoming, protocol-specific status messages from the device and turn them into normalized “events”. It is also responsible for accepting normalized commands (such as “on” and “off”) and turning those into the protocol-specific commands that can be sent to the device to affect the desired action.

For example, for a Z-Wave compatible on-off switch, the incoming status messages used by the device to report an “on” or “off” state are as shown below:

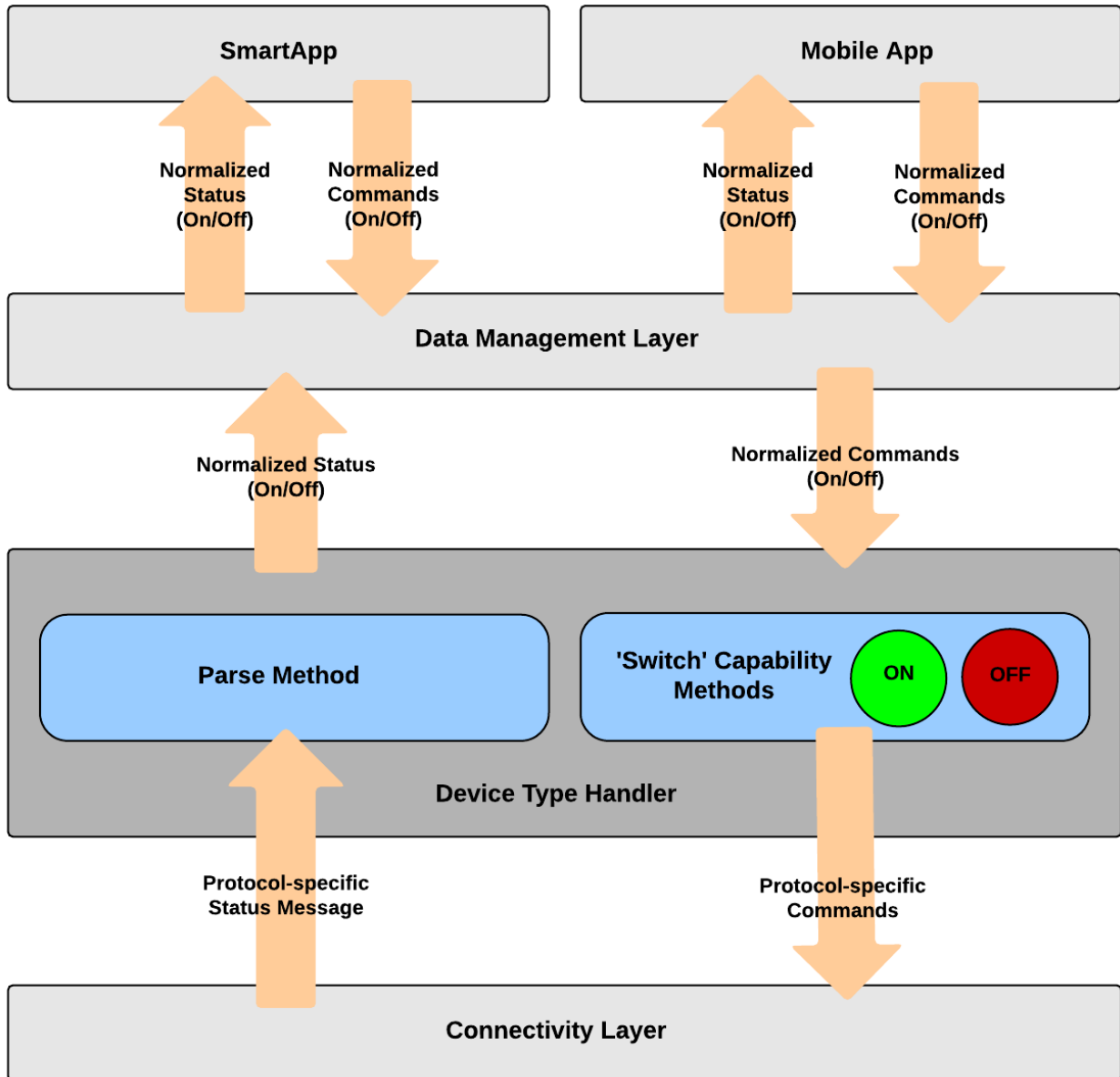
Device Command	Protocol-Specific Command Message
on	command: 2003, payload: FF
off	command: 2003, payload: 00

Whereas the device status reported to the SmartThings platform for the device is literally just a simple “on” or “off”.

Similarly, when a SmartApp or the mobile app invoked an “on” or “off” command for a switch device, the command that is sent to the Device Handler is just that simple: “on” or “off”. The Device Handler must turn that simple command into a protocol-specific message that can be sent down to the device to affect the desired action.

The table below shows the actual Z-Wave commands that are sent to a Z-Wave switch by the Device Handler.

Device Command	Protocol-Specific Command Message
On	2001FF
Off	200100



130.1 Core concepts

To understand how device handlers work, a few core concepts need to be discussed.

130.1.1 Capabilities

Capabilities are the interactions that a device allows. They provide an abstraction layer that allows SmartApps to work with devices based on the capabilities they support, and not be tied to a specific manufacturer or model.

Consider the example of the “Switch” capability. In simple terms, a switch is a device that can turn on and off. It may be that a switch in the traditional sense (for example an in-wall light switch), a connected bulb (a Hue or Cree bulb), or even a music player. All of these unique devices have a Device Handler, and those Device Handler’s support the “Switch” capability. This allows SmartApps to only require a device that supports the “Switch” capability and thus work with a variety of devices including different manufacturer and model-specific “switches”. The SmartApp can then interact with the device knowing that it supports the “on” and “off” command (more on commands below), without caring about the specific device being used.

This code illustrates how a SmartApp might interact with a device that supports the “Switch” capability:

```
preferences() {
    section("Control this switch"){
        input "theSwitch", "capability.switch", multiple: false
    }
}

def someEventHandler(evt) {
    if (someCondition) {
        theSwitch.on()
    } else {
        theSwitch.off()
    }

    // logs either "switch is on" or "switch is off"
    log.debug "switch is ${theSwitch.currentSwitch}"
}
```

The above example illustrates how a SmartApp requests a device that supports the “Switch” capability. When installing the SmartApp, the user will be able to select any device that supports the “Switch” capability - be it an in-wall light switch, a connected bulb, a music player, or any other device that supports the “Switch” capability.

The *Capabilities Reference* (page 655) outlines all the supported capabilities.

Device Handlers typically support more than one capability. A Device Handler for a Hue bulb would support the “Switch” capability as well as the “Color Control” capability. This allows SmartApps to be written in a very flexible manner.

Commands and attributes deserve their own discussion - let’s dive in.

130.1.2 Commands

Commands are the actions that your device can do. For example, a switch can turn on or off, a lock can lock or unlock, and a valve can open or close. In the example above, we issue the “on” and “off” command on the switch by invoking the `on()` or `off()` methods.

Commands are implemented as methods on the Device Handler. When a device supports a capability, it is responsible for implementing all the supported command methods.

130.1.3 Attributes

Attributes represent particular state values for your device. For example, the switch capability defines the attribute “switch”, with possible values of “on” and “off”.

In the example above, we get the value of the “switch” attribute by using the “current<attributeName>” property (`currentSwitch`).

Attribute values are set by creating Events where the attribute name is the name of the Event, and the attribute value is the value of the Event. This is discussed more in the Parse and Events documentation

Like commands, when a device supports a capability, it is responsible for ensuring that all the capability’s attributes are implemented.

130.1.4 Actuator and Sensor

If you look at the *Capabilities Reference* (page 655) , you’ll notice two capabilities that have no attributes or commands - “Actuator” and “Sensor”.

These capabilities are “marker” or “tagging” capabilities (if you’re familiar with Java, think of the Cloneable interface - it defines no state or behavior).

The “Actuator” capability defines that a device has commands. The “Sensor” capability defines that a device has attributes.

If you are writing a Device Handler, it is a best practice to support the “Actuator” capability if your device has commands, and the “Sensor” capability if it has attributes. This is why you’ll see most Device Handlers supporting one of, or both, of these capabilities.

The reason for this is convention and forward-looking abilities - it can allow the SmartThings platform to interact with a variety of devices if they *do* something (“Actuator”), or if they report something (“Sensor”).

130.2 Protocols

SmartThings currently supports both the *Z-Wave* and *ZigBee* wireless protocols.

Since the Device Handler is responsible for communicating between the device and the SmartThings platform, it is usually necessary to understand and communicate in whatever protocol the device supports. This guide will discuss both *Z-Wave* and *ZigBee* protocols at a high level.

130.3 Execution location

With the original SmartThings Hub, all Device handlers execute in the SmartThings cloud. With the new Samsung SmartThings Hub, certain Device handlers may run locally on the Hub or in the SmartThings cloud. Execution location varies depending on a variety of factors, and is managed by the SmartThings internal team.

As a SmartThings developer, you should write your Device Handlers to satisfy their specific use cases, regardless of where the handler executes. There is currently no way to specify or force a certain execution location.

Simulator

Using the IDE Simulator, we can model the behavior of the device without actually requiring a physical device.

131.1 Overview

On the right-hand side of the IDE, after you install a Device Handler, you'll see the Simulator. The image below is the Simulator seen after installing the "Z-Wave Switch" Device Handler (available via the *Browse Device Templates* menu).

Go ahead, try it out. Install the Device Handler in the IDE, and choose a virtual switch. Modify some of the Simulator metadata as you read through this and see what happens.

The purpose of the Simulator metadata is to model the behavior of the physical device. Using the Simulator, we can test sending messages and commands to our Device Handler.

There are two types of Simulator declarations to define in a Device Handler - "status" and "reply".

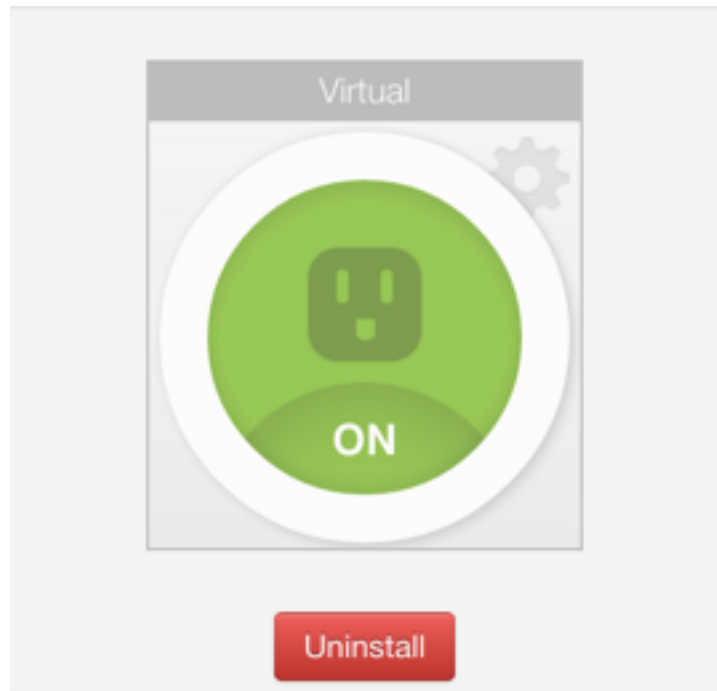
131.2 Status

The "status" declarations specify actions that result in a person physically actuating the device. In the case of the Z-Wave switch, for example, we have:

```
status "on": "command: 2003, payload: FF"
status "off": "command: 2003, payload: 00"
```

`status()` takes a map as an argument. The key ("on" in the example above) is just a name for the action. The value ("command: 2003, payload: FF") is the message that the device will send to the Device Handler's `parse(message)` method when that action is taken on the physical device.

In the Simulator, each status key ("on" or "off" in the example above) will be an available message in the Simulator.



Tools

Messages

off

Raw

Commands

Switch

Polling

Refresh

Indicator

131.3 Reply

The “reply” declarations specify responses that the physical device will send to the Device Handler when it receives a certain message from the Hub. For a Z-Wave switch, for example, we specify:

```
reply "2001FF,delay 100,2502": "command: 2503, payload: FF"
reply "200100,delay 100,2502": "command: 2503, payload: 00"
```

Just like `status()`, `reply()` accepts a map as a parameter. The key is a comma-separated list of the raw commands sent to the device, i.e. what’s returned from the Device Handler’s command methods. For example, the Z-Wave switch commands that send the above methods are:

```
def on() {
    delayBetween ([
        zwave.basicV1.basicSet (value: 0xFF).format (),
        zwave.switchBinaryV1.switchBinaryGet ().format ()
    ])
}

def off() {
    delayBetween ([
        zwave.basicV1.basicSet (value: 0x00).format (),
        zwave.switchBinaryV1.switchBinaryGet ().format ()
    ])
}
```

Those methods will return the values in the first arguments of the reply declarations. The second argument in the reply declarations works the same way as the status declarations - they define messages sent to the parse method. But in this case it’s in response to commands, not physical actuations.

131.4 Summary

The purpose of these declarations is to allow a virtual device to function in the IDE Simulator, without being attached to a physical device. The `status()` method allows us to simulate physical actuation, while the `reply()` method allows us to simulate sending messages to the device in response to a command from the Hub.

Definition

The definition metadata defines core information about your Device Handler. The initial values are set from the values entered when creating your Device Handler.

Example definition metadata:

```
metadata {
  definition(name: "test device", namespace: "yournamespace", author: "your name") {
    capability "Alarm"
    capability "battery"

    attribute "customAttribute", "string"

    command "customCommand"

    fingerprint profileId: "0104", inClusters: "0000,0003,0006",
      outClusters: "0019"
  }
  ...
}
```

The definition method takes a map of parameters, and a closure.

The supported parameters are:

name The name of the Device Handler.

namespace The namespace for this Device Handler. This should be your github user name. This is used when looking up Device Handlers by name to ensure the correct one is found, even if someone else has used the same name.

author The author of this Device Handler.

The closure defines the capabilities, attributes, commands, and fingerprint information for your Device Handler.

132.1 Capabilities

To define that your device supports a capability, simply call the `capability` method in the closure passed to `definition`.

The argument to the `capability` method is the Capability name.

```
capability "Actuator"  
capability "Power Meter"  
capability "Refresh"  
capability "Switch"
```

132.2 Attributes

If you need to define a custom attribute for your Device Handler, call the `attribute()` method in the closure passed to the `definition()` method:

`attribute(String attributeName, String attributeType, List possibleValues = null)`

attributeName Name of the attribute.

attributeType Type of the attribute. Available types are “string”, “number”, and “enum”.

possibleValues Optional. The possible values for this attribute. Only valid with the “enum” attributeType.

```
// String attribute with name "someName"  
attribute "someName", "string"  
  
// enum attribute with possible values "light" and "dark"  
attribute "someOtherName", "enum", ["light", "dark"]
```

132.3 Commands

To define a custom command for your Device Handler, call the `command()` method in the closure passed to the `definition()` method:

`command(String commandName, List parameterTypes = [])`

commandName The name of the command. You must also define a method in your Device Handler with the same name.

parameterTypes Optional. An ordered list of the parameter types for the command method, if needed.

```
// command name "myCommand" with no parameters  
command "myCommand"  
  
// comand name myCommandWithParams that takes a string and a number parameter  
command "myCommandWithParams", ["string", "number"]  
  
...  
  
// each command specified in the definition must have a corresponding method  
  
def myCommand() {  
    // handle command  
}  
  
// this command takes parameters as defined in the definition  
def myCommandWithParams(stringParam, numberParam) {  
    // handle command  
}
```


132.4 Fingerprinting

When a ZigBee or Z-Wave device is added to the SmartThings Hub, we need a way to determine which device type to assign it. This process is known as a “join” process, or “fingerprinting”.

Device Handlers define “fingerprints” to specify which devices or what kinds of devices they support. Then, when a device is added, its join information is compared to all fingerprints in the default handlers and your self-published handlers to determine which type of device it is.

The fingerprinting process differs between ZigBee and Z-Wave devices.

132.4.1 ZigBee fingerprinting

For ZigBee devices, the main profileIds you will need to use are:

- HA: Home Automation (0104)
- SEP: Smart Energy Profile
- ZLL: ZigBee Light Link (C05E)

The input and output clusters are defined specifically by your device and should be available via the device’s documentation.

An example of a ZigBee fingerprint definition:

```
fingerprint profileId: "C05E", inClusters: "0000,0003,0004,0005,0006,0008,0300,1000", outClusters: "0
```

You can also include the manufacturer and model name in the fingerprint to limit the fingerprint to a specific product:

```
fingerprint inClusters: "0000,0001,0003,0020,0406,0500", manufacturer: "NYCE", model: "3014"
```

132.4.2 Z-Wave fingerprinting

Z-Wave fingerprints used to be based on the format used for ZigBee, but there is now a new format that is preferred. You may see the original fingerprints on older Device Handlers; see below for information on the legacy format.

The best place to start is to add your device to SmartThings and look for the *Raw Description* in its details view in the SmartThings developer tools.

Z-Wave raw description

Z-Wave devices added since the introduction of the new format will have raw description strings with multiple key-value fields, such as:

```
zw:zs type:2101 mfr:0086 prod:0102 model:0064 ver:1.04 zwv:4.05 lib:03 cc:55,86,72,98,84 out:5A see
```

Not all fields will be present for every device.

zw: This field will start with ‘L’ for listening devices, ‘S’ for sleepy devices, and ‘F’ for beamable devices. See the *Z-Wave Primer* (page 507) for the meaning of those terms. That capital letter will be followed by a lowercase ‘s’ if the device is securely included into the network via the Z-Wave Security Layer.

type: This field is the Z-Wave Device Class as a 16-bit hexadecimal number that combines the Generic and Specific Device Class codes. ¹

mfr: This 16-bit hexadecimal number identifies the device manufacturer. ¹ The three values of `mfr`, `prod` and `model` uniquely identify a certified Z-Wave product.

prod: This 16-bit hexadecimal number is the Product Type ID reported by the device.

model: This 16-bit hexadecimal number is the Product ID reported by the device.

ver: This is the application firmware version reported by the device.

zwv: This is the version of the Z-Wave protocol stack being used by the device.

lib: This indicates the type of Z-Wave protocol library the device is based on. '01' is a static controller, '02' is a remote controller, '07' is a bridge controller, and other values are normal non-controller devices.

cc: The list of Z-Wave command classes supported by the device (without security encapsulation). See the [Z-Wave Command Reference](#) for the command classes represented by each hex code.

ccOut: The list of Z-Wave command classes that the device can control. This refers to commands sent to other devices versus reports generated by the device.

sec: These command classes are supported by the device only via Z-Wave Security encapsulation.

secOut: These command classes are *controlled* by the device only via Z-Wave Security encapsulation.

role: This indicates the Z-Wave Plus Role Type. ¹

ff: This stands for “form factor” and corresponds to the Z-Wave+ Installer Icon type (An offset of 0x8000 is added for implementation reasons). ¹

ui: This corresponds to the Z-Wave+ User Icon type.

New Z-Wave fingerprint format

If you're writing a Device Handler for a specific device, you can base the fingerprint on the manufacturer info. For example, the fingerprint to match the raw description example above would be:

```
fingerprint mfr: "0086", prod: "0102", model: "0064"
```

No other parameters are required. Note that you need to add quotes and commas to the more concise raw description format to make it valid Groovy code.

Sometimes related products are grouped under the same 'prod' ID. In that case you can use a fingerprint without the 'model' parameter.

If you are writing a general Device Handler that supports all devices of a certain type, you can still base the fingerprint on command class support.

```
fingerprint type: "10", cc: "25,32"
```

That fingerprint would match all devices of the Binary Switch generic device class – i.e. their 'type' starts with “10” – that support the Binary Switch (0x25) and Meter (0x32) command classes.

The supported parameters are:

type: Matches if it's equal to or a prefix of the device's 'type' value in the raw description. Aliased as 'deviceId'.

mfr, prod, model: Matches if 'mfr' matches the raw description and 'prod' and 'model' match as prefixes (if present).

¹ See [this document](#) for the values of identifiers defined by the Z-Wave standard.

cc, ccOut: Takes a list of command class codes as a string: comma-separated, uppercase hexadecimal. Matches if all listed command class codes are reported as supported or controlled respectively in the device’s raw description.

sec, secOut: The same as the previous parameter, but only matches against command classes the device supports/controls only via Z-Wave Security encapsulation.

ff/ui: Either of these parameters can be used to match against the corresponding fields of the raw description. It is only possible to use one of the following in a single fingerprint: ‘type’, ‘deviceId’, ‘ff’, ‘ui’.

deviceJoinName: Not used for matching. If the fingerprint matches, the device will appear to the user with this name.

When multiple device fingerprints match an added Z-Wave device, they are ranked first by number of ‘mfr’, ‘prod’, and ‘model’ parameters, then by the number of command classes listed, and finally by the length of the ‘type’, ‘ff’, or ‘ui’ parameter. When fingerprints have the same rank, self-published Device Handlers take precedence over the default production ones.

Legacy Z-Wave fingerprint format

Legacy fingerprints include the device class – or type value (see above) – in the `deviceId` parameter and the command classes it supports in the `inClusters` parameter. So the fingerprint:

```
fingerprint deviceId:"0x1104", inClusters:"0x26, 0x2B, 0x2C, 0x27, 0x73, 0x70, 0x86, 0x12", outClusters:"0x05"
```

would be formatted in the new style as:

```
fingerprint type: "1104", cc: "26,2B,2C,27,73,70,86,72", ccOut: "20"
```

132.4.3 Fingerprinting best practices

Add multiple fingerprints

A Device Handler can have multiple fingerprints in order to work with multiple versions of a device. Each fingerprint is independent. If any of them is the highest ranking match, the device will use your device type.

You can distinguish between the different devices that use the handler by adding the ‘deviceJoinName’ parameter. For example:

```
fingerprint profileId: "0104", inClusters: "0000, 0003, 0004, 0005, 0006, 0008, 0702"
fingerprint profileId: "0104", inClusters: "0000, 0003, 0004, 0005, 0006, 0008, 0702, 0B05", outClusters: "0B05"
fingerprint profileId: "0104", inClusters: "0000, 0003, 0004, 0005, 0006, 0008, 0702, 0B05", outClusters: "0B05"
fingerprint profileId: "0104", inClusters: "0000, 0003, 0004, 0005, 0006, 0008, 0702, 0B05", outClusters: "0B05"
```

If an added device supports the `inClusters` in the first fingerprint but doesn’t match all the extra info in any of the next three, it will join with the name from the handler’s definition metadata, in this case “ZigBee Dimmer Power.”

Device pairing process

The order of the `inClusters` and `outClusters` lists is not important to the pairing process. It is a best practice, however, to list the clusters in ascending order.

The device can have more clusters than the fingerprint specifies, and it will still pair. If one of the clusters specified in the fingerprint is incorrect, the device will *not* pair.

Overly general fingerprints

If you wish to publish or share a Device Handler, you must make sure that the fingerprints do not capture other devices that aren't covered by your handler.

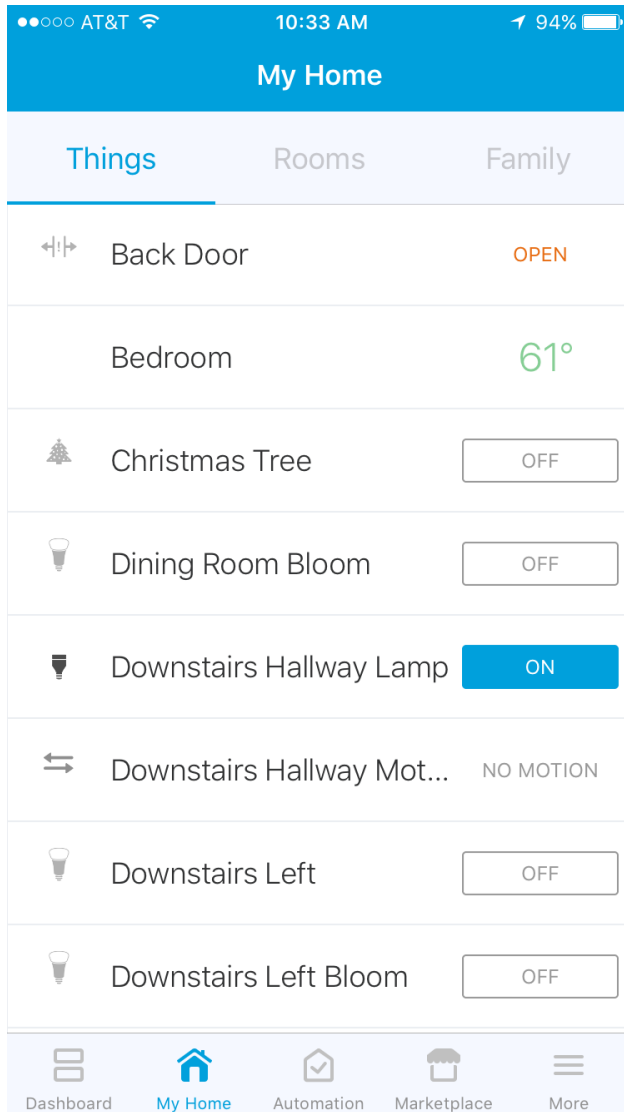
If you copied a working fingerprint from a default or template handler, it would be ambiguous which type should match if yours was published. The easiest way to remedy this is to include manufacturer and model info in all fingerprints.

Tiles

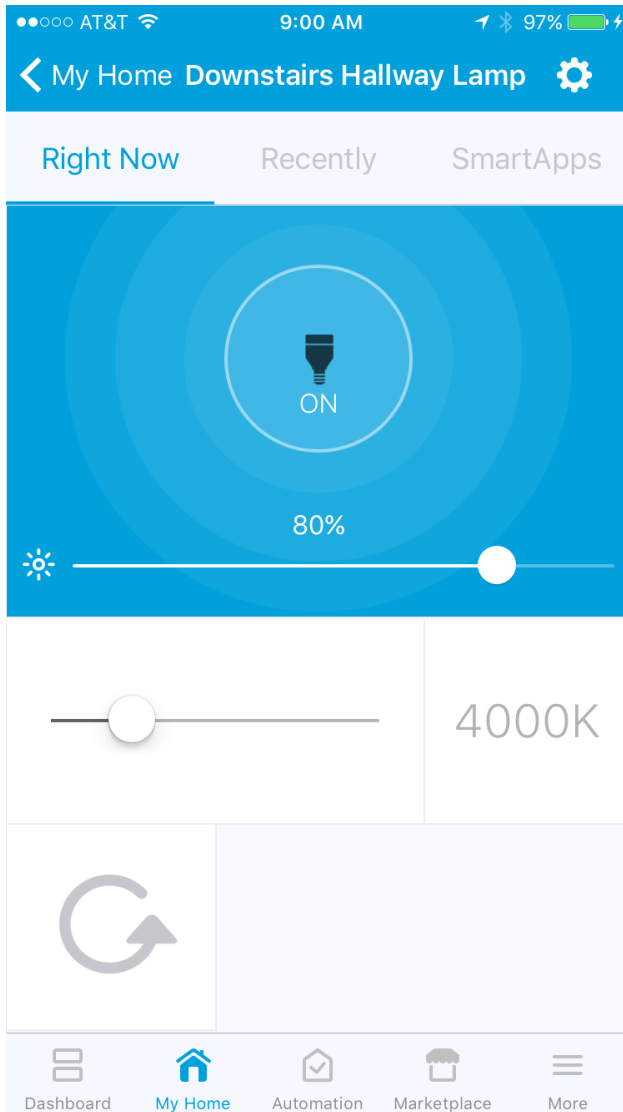
Tiles define how devices are visually represented in the SmartThings mobile application. Every Device Handler specifies how the device will appear in the mobile application by specifying one or more tiles.

133.1 Overview

When a user goes to the Things view in the mobile app, they see all their devices listed:



When tapping on one of the devices in the Things view, the user will see the Details view for that device:



The Details view is where a user can get comprehensive information about the device, as well as actuate the device (if applicable) by interacting with the tile.

A tile may be display only, or can be configured to perform an action on the device when interacted with.

Each tile is associated with one or more attributes of the device. Some tiles are display-only, while others allow the user to interact with the tile to actuate the device.

133.2 Tiles basics

Tip: If you're the type that prefers to view and experiment with real code examples as you learn, check out the *Examples* (page 497).

Tiles definition and layout is specified using the `tiles()` builder in the Device Handler's metadata, and looks like this (we'll dive into the details shortly):

```

metadata {
    definition (
        ...
    )
}

tiles(scale: 2) {
    // standard tile with actions named
    standardTile("switch", "device.switch", width: 2, height: 2, canChangeIcon: true) {
        state "off", label: '${currentValue}', action: "switch.on",
            icon: "st.switches.switch.off", backgroundColor: "#ffffff"
        state "on", label: '${currentValue}', action: "switch.off",
            icon: "st.switches.switch.on", backgroundColor: "#00a0dc"
    }

    // value tile (read only)
    valueTile("power", "device.power", decoration: "flat", width: 2, height: 2) {
        state "power", label: '${currentValue} Watts'
    }

    // the "switch" tile will appear in the Things view
    main("switch")

    // the "switch" and "power" tiles will appear in the Device Details
    // view (order is left-to-right, top-to-bottom)
    details(["switch", "power"])
}
}

```

It's important to understand that tiles configuration is part of the device's static metadata. When the SmartThings platform executes the `tiles()` builder you have defined, it doesn't yet know anything about the actual device or the current device state. Only later, when the device details screen is rendered in the mobile client, does the platform know information about the specific device. For this reason, trying to conditionally configure tiles based on device state will not work.

Tiles are associated with attributes of a device. Device tiles come in two varieties:

1. Single-attribute tiles. These tiles are associated with one attribute of the device.
2. Multi-attribute tiles. These tiles can display information about multiple attribute of a device.

133.2.1 Main and details tiles configuration

The main tile is what appears in the Things view. It's configured in the `tiles()` builder with `main()`:

```

tiles(scale: 2) {
    standardTile(name: 'someTile', ...)
    controlTile(name: 'otherTile', ...)

    // tile with name 'someTile' appears in the Things view
    main('someTile')
}

```

Use `details()` to specify all other tiles that should be available on the device details screen. The tiles will layout in left-to-right, top-to-bottom order beginning with the first argument:

```

tiles(scale: 2) {
    standardTile(name: 'someTile', ...)
    controlTile(name: 'otherTile', ...)
}

```



```
valueTile(name: 'valueTile', ...)

main('someTile')
// someTile is top left, then otherTile, then anotherTile,
// all flowing left-to-right, top-to-bottom:
details('someTile', 'otherTile', 'anotherTile')
}
```

133.2.2 Grid layout

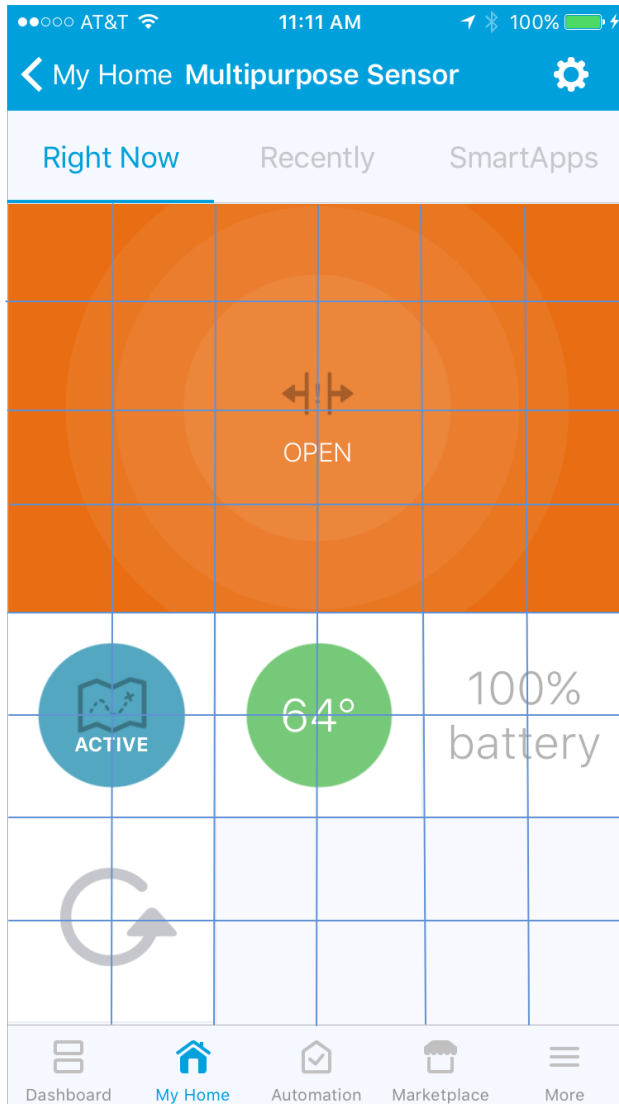
Tiles are rendered using a grid layout. Tiles support either a **6 x Unlimited** (6 wide, unlimited height) or **3 x Unlimited** (3 wide, unlimited height) layout. The grid system used is controlled by the `scale` argument to the `tiles` builder. A value of 1 (the default) enables the *3 x Unlimited* grid; a value of 2 enables the *6 x Unlimited* grid:

```
// 3 x Unlimited grid
tiles(scale: 1) {...}

// 6 x Unlimited grid
tiles(scale: 2) {...}
```

SmartThings recommends using the *6 x Unlimited* layout, as it offers a more attractive user experience. Older versions of the SmartThings mobile application that do not support the *6 x Unlimited* layout will be scaled back.

Here you can see how the tiles defined above are laid out using the *6 x Unlimited* grid (using the `scale: 2` option):



133.2.3 Tile size

Every tile can specify a `width` and a `height`, which controls the size of the tile within the grid layout. If not specified, the tile will default to a width and height of 1.

133.2.4 Allowing the user to change the icon

We can specify the `canChangeIcon: true` option to allow the user to select an icon of their choosing when editing the device:

```
standardTile("switch", "device.switch", width: 2, height: 2, canChangeIcon: true) {...}
```

If not specified, `canChangeIcon` is assumed to be `false`. Only the tile specified as the main tile should specify `canChangeIcon`.

133.3 Tiles and Attribute state

Tiles display data about a device’s attributes, and may allow those attributes to be updated through user interaction.

Let’s explore how this works by considering an example. Consider the case of a Switch - it could be a smart outlet, an in-wall switch, or a smart bulb. Regardless of the specific device, we want to display a tile that shows the current state of the switch (on or off), and allows the user to toggle the switch by pressing the tile. We accomplish this by associating one or more states for a tile definition.

When we define a tile, we associate it with a specific attribute of the device. In our Switch example, this would be the “switch” attribute of the switch capability:

```
standardTile("tileName", "device.switch", width: 2, height: 2) {...}
```

Now that we’ve associated the tile with the switch attribute, we need to configure how it will display for the attribute’s possible states. For single-attribute tiles (`standardTile` is a single-attribute tile), we do this using `state`. Multi-attribute tiles use `attributeState`, which is used in the same way.

For attributes that have a finite, discrete set of possible values (for example, “on” or “off”, “wet” or “dry”, “open” or “closed”), we create a `state` definition for each possible value. Each `state` definition can be configured to customize the display and what should happen (if anything) when the tile is pressed by the user. For attributes whose value are not finite values (examples include “temperature”, “power”, or the “level” of a dimmable switch), we simply use one `state` for the attribute:

```
valueTile("tileName", "device.level", width: 2, height: 2) {
    state "level", label: '${currentValue}'
}
```

You can learn more about using dynamic state labels ('`${currentValue}`' above) [here](#) (page 480).

In the case of the “switch” attribute, we need to define two states, one for “on” and one for “off”:

```
standardTile("tileName", "device.switch", width: 2, height: 2) {
    state "off", label: "off", icon: "st.switches.switch.off", backgroundColor: "#ffffff"
    state "on", label: "on", icon: "st.switches.switch.on", backgroundColor: "#00a0dc"
}
```

The above tile definition is pretty self-explanatory. When the “switch” attribute is “off”, the label of the tile will be “off”, the icon will be “st.switches.switch.off”, and the background color will be white (#ffffff). It’s similarly easy to understand how the tile will appear when the switch is “on”.

133.3.1 State actions

Tile states can define what should happen when the tile is interacted with by specifying an `action`. For example, to allow a switch to be toggled when pressed, we specify what should happen for each attribute state:

```
standardTile("tileName", "device.switch", width: 2, height: 2) {
    state "off", label: "off", icon: "st.switches.switch.off", backgroundColor: "#ffffff", action: "switch.off"
    state "on", label: "on", icon: "st.switches.switch.on", backgroundColor: "#00a0dc", action: "switch.on"
}
```

The value of the `action` can be formatted in one of two ways:

1. In the form "`<capability>.<command>`".
2. In the simpler form "`<command>`". This form is required for custom (non-capability) commands.

We are showing the form "`<capability>.<command>`" form above, which translates to `action: "switch.on"`. We could also simply specify the command, which would look like: `action: "on"`.

If you're curious about commands that take parameters (`on()` and `off()` do not), you do **not** need to specify parameters in the `action`. Any parameters will be populated and passed to the command method by the specific tile control.

Note: While both action forms are supported, you'll most frequently see the form "`<capability>.<command>`" in Device Handlers. This form can be somewhat confusing when the capability has a space in its name; consider this example that would call the `setLevel` command on a "Switch Level" capability:

```
action: "switch_level.setLevel"
```

The above reads awkwardly for many, and can cause confusion.

Because of this, we prefer the short form of `action: "<command>".`

133.3.2 Transition states

We can use the `nextState` option in `state` (single-attribute tiles) or `attributeState` (Multi-Attribute Tiles) to show that the device is transitioning to a next state. This is useful to provide visual feedback that the device state is transitioning. When the attribute's state does change, the tile will be updated according to the state defined for that attribute.

To define a transition state, simply define a `state` for the transition, and reference that state using the `nextState` option.

Here's an example that uses a transition state for the "switch" attribute:

```
standardTile("switch", "device.switch", width: 2, height: 2) {
    state "off", label: 'Off', action: "switch.on", icon: "st.switches.switch.off", backgroundColor: "#ffffff"
    state "on", label: 'On', action: "switch.off", icon: "st.switches.switch.on", backgroundColor: "#00a0dc"
    state "turningOn", label: 'Turning on', icon: "st.switches.switch.on", backgroundColor: "#00a0dc", nextState: "on"
    state "turningOff", label: 'Turning off', icon: "st.switches.switch.off", backgroundColor: "#ffffff", nextState: "off"
}
```

133.3.3 State labels

We can hard-code a label for state values, or use the state name or current value of the attribute. The following label values can be used to display real-time information about the device:

Label	Description
<code>label: '\${currentValue}'</code>	The current value of this attribute's state. This is used when the attribute doesn't have a discrete value set, like temperature or power.
<code>label: '\${name}'</code>	The name of the attribute state. This is useful when the attribute state is a discrete value, like "on" or "off".

Here's an example of using the state name as the label:

```
standardTile("switch", "device.switch") {
    // use the state name as the label ("off" and "on")
    state "off", label: '${name}', action: "switch.on", icon: "st.switches.switch.off", backgroundColor: "#ffffff"
    state "on", label: '${name}', action: "switch.off", icon: "st.switches.switch.on", backgroundColor: "#00a0dc"
}
```

When using the current attribute value, the attribute value must be set by sending an Event. For simplicity, the code examples in this documentation typically will not show the attribute value being set. Just know that if a label is set like this:

```
valueFile("power", "device.power") {
    // label will be the current value of the power attribute
    state "power", label: '${currentValue} W'
}
```

The Device Handler needs to send an Event for the "power" attribute somewhere:

```
sendEvent(name: "power", value: 42)
```

Important: Dynamic device state values like `'${currentValue}'` and `'${name}'` **must be used inside single quotes**. This is in contrast to Groovy's string interpolation that requires double quotes.

This is required because when the platform executes the `tiles()` builder, it doesn't know anything about the actual device yet. Using single quotes will allow the platform to manually substitute the actual value when the device is rendered on the mobile app.

133.3.4 Background color

We've seen in the examples above that states can be configured to appear a certain color using `backgroundColor`. The value to the `backgroundColor` option is a hexadecimal value of the color.

We can also specify an array of background colors for attribute values that fall along a range. This allows for greater user feedback for a given attribute value, since we can specify the background color for various values. When the value is between the specified ranges, the resulting color will be a shade between the two specified colors. The "temperature" attribute is a common example of this. It's typical to see a tile definition for temperature like this:

```
valueFile("temperature", "device.temperature", width: 2, height: 2) {
    state("temperature", label: '${currentValue}', unit: "dF",
        backgroundColors: [
            [value: 31, color: "#153591"],
            [value: 44, color: "#1e9cbb"],
            [value: 59, color: "#90d2a7"],
            [value: 74, color: "#44b621"],
            [value: 84, color: "#f1d801"],
            [value: 95, color: "#d04e00"],
            [value: 96, color: "#bc2323"]
        ]
    )
}
```

The argument to `backgroundColors` is a list of maps, where each map specifies the hexadecimal color a specific value. When the attribute value matches a value specified, the color specified will be the background color of the tile. When the value is between two specified values, the color will be a linear interpolation between the specified ranges.

In the example above, we defined that at 84 degrees the background color will be a shade of green ("44b621"). When the temperature reaches 95 degrees, the color will be a shade of yellow ("f1d801"). When the temperature is between 84 and 95 degrees, the background color will be between green and yellow. Increasing the temperature causes the color to become progressively more yellow, until arriving at 95 degrees. Similarly, decreasing the temperature causes the color to become more and more green, until arriving at 84 degrees.

Once an upper or lower bound has been reached, the background color will no longer change. In the example above, that means that decreasing the temperature below 31 degrees or above 96 degrees will not cause the background color to change from the colors specified at those values.

133.3.5 State selection algorithm

The following algorithm is used to determine which state to display, when there are multiple states:

1. If a state is defined for the attribute's current value, it will render that.
2. If no state exists for the attribute value, it will render a state that has specified `defaultState: true`. Use this in place of the “default” state name that you may see in some Device Handlers.
3. If no state matches the above rules, it will render the first state declaration.

133.3.6 Icons

A tile's state may specify an icon to render using the `icon` option:

```
tileAttribute ("device.power", key: "SECONDARY_CONTROL") {
    attributeState "power", label: '${currentValue}W', icon: "st.Appliances.appliances17"
}
```

We can use an icon provided by SmartThings as above, or an accessible URL to an icon.

Note: Using icons is [discussed frequently](#) in the SmartThings developer community forums.

133.4 Single-Attribute Tiles

Single-attribute tiles are associated with a single device attribute. There are several different single-attribute tiles available for use, as documented below.

133.4.1 Standard Tile

Use a Standard Tile for attributes that have discrete, specific values. For example, a switch is either “on” or “off”; a moisture sensor is “wet” or “dry”; a contact sensor is “open” or “closed”.

Here's a standard tile that shows if a switch is on or off.

```
standardTile("actionFlat", "device.switch", width: 2, height: 2, decoration: "flat") {
    state "off", label: '${currentValue}', action: "switch.on", icon: "st.switches.switch.off", background: "#ffffff"
    state "on", label: '${currentValue}', action: "switch.off", icon: "st.switches.switch.on", background: "#ffffff"
}
```

The above tile definition would render as (when the switch is on):

Standard Tiles may be styled with a ring (the default), or flat, by using the `decoration` option:

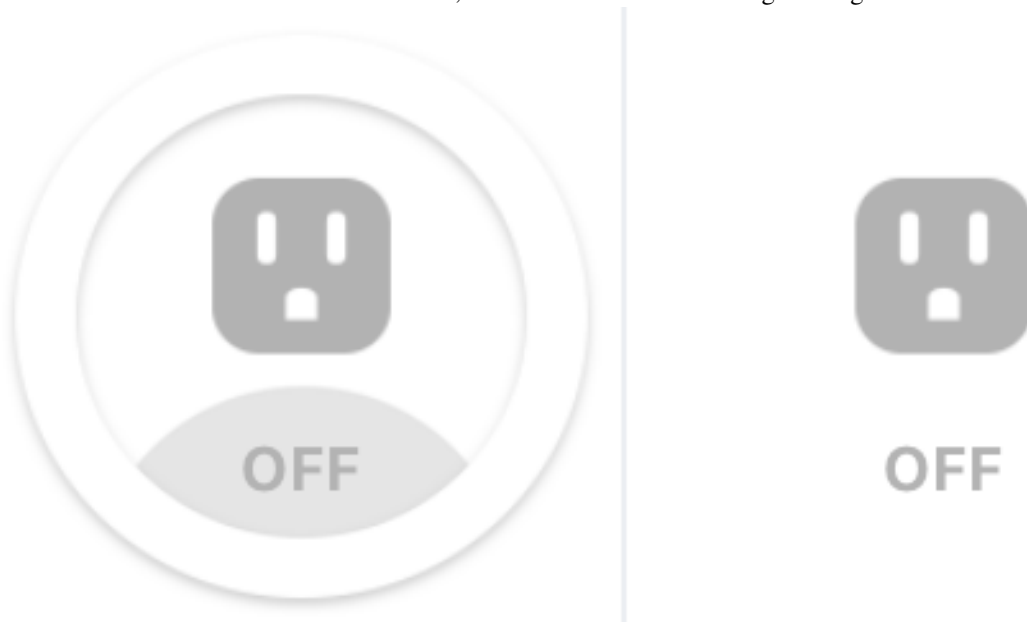
```
// standard tile with actions
standardTile("actionRings", "device.switch", width: 2, height: 2, canChangeIcon: true) {
    state "off", label: '${currentValue}', action: "switch.on", icon: "st.switches.switch.off", background: "#ffffff"
    state "on", label: '${currentValue}', action: "switch.off", icon: "st.switches.switch.on", background: "#ffffff"
}

// standard flat tile without actions
standardTile("noActionFlat", "device.switch", width: 2, height: 2, canChangeIcon: true) {
    state "off", label: '${currentValue}', icon: "st.switches.switch.off", backgroundColor: "#ffffff"
}
```



```
state "on", label: '${currentValue}', icon: "st.switches.switch.on", backgroundColor: "#00a0dc"
}
```

The above tiles definition renders as below, with the tile on the left being the ring decoration:



Tip: Check out the [Examples](#) (page 497) to see it in action!

133.4.2 Value Tile

Use a Value Tile for attributes that have non-discrete values. Typical examples include temperature, humidity, or power values.

The following shows a few examples of the Value Tile:

```
tiles(scale: 2) {
    valueTile("integerFloat", "device.integerFloat", width: 2, height: 2) {
        state "val", label: '${currentValue}', defaultState: true
    }

    valueTile("pi", "device.pi", width: 2, height: 2) {
        state "val", label: '${currentValue}', defaultState: true
    }




    valueTile("floatAsText", "device.floatAsText", width: 2, height: 2) {
        state "val", label: '${currentValue}', defaultState: true
    }

    valueTile("bgColor", "device.integer", width: 2, height: 2) {
        state "val", label: '${currentValue}', backgroundColor: "#e86d13", defaultState: true
    }

    valueTile("bgColorRange", "device.integer", width: 2, height: 2) {
        state "val", label: '${currentValue}', defaultState: true, backgroundColors: [
            [value: 10, color: "#ff0000"],
            [value: 90, color: "#0000ff"]
        ]
    }
}

def installed() {
    sendEvent(name: "integer", value: 47)
    sendEvent(name: "integerFloat", value: 47.0)
    sendEvent(name: "pi", value: 3.14159)
    sendEvent(name: "floatAsText", value: "3.14159")
}
```

This renders as:

47.0	3.14159	3.14159
		

Note: While it's possible to specify an action for a Value Tile, that is not the intended purpose. If your tile should support an action, use a Standard Tile. Value Tiles are intended to be used for display-only attributes.

Tip: Check out the *Examples* (page 497) to see it in action!

133.4.3 Slider Control Tile

Use a Slider Control Tile to display a tile that shows a value along a range, and allows the user to adjust the value using the slider control.

These tiles are useful for attributes like the level of a dimmable bulb.

Here's an example of a Slider Control Tile:

```
controlTile("levelSliderControl", "device.level", "slider",
    height: 1, width: 2) {
    state "level", action:"switch level.setLevel"
}
```

This renders as:



By default, the range of the slider will be 0-100. You can specify a custom range by using a `range` parameter. It is a string, in the form "`(<lower bound>..<upper bound>)`". Only integers (negative and positive) are supported for custom ranges; decimal values will not work.

```
controlTile("levelSliderControl", "device.level", "slider", height: 1,
    width: 2, inactiveLabel: false, range:"(20..80)") {
    state "level", action:"switch level.setLevel"
}
```

Tip: Check out the [Examples](#) (page 497) to see it in action!

133.4.4 Color Control Tile

If your device supports the `colorControl` capability, you can use a Control Tile that displays a color wheel. The user can then set the color by interacting with the control.

Here's an example of a color control tile:

```
controlTile("rgbSelector", "device.color", "color", height: 6, width: 6,
    inactiveLabel: false) {
    state "color", action:"color control.setColor"
}
```

The tile may render differently depending on the mobile OS. The command method specified by `action` will be called with a map that looks like this:

```
[red:241, hex:#F1E3FF, saturation:10.980392, blue:255, green:227, hue:75.0]
```

The values are summarized in the table below:

Key	Description
red	The red value chosen in the standard RGB color space
hex	The hexadecimal representation of the color chosen
saturation	The saturation value of the value chosen, between 0 and 100
blue	The blue value chosen in the standard RGB color space
green	The green value chosen in the standard RGB color space
hue	The hue value of the color chosen, between 0 and 100

You may also see a `level` and `alpha` attribute returned from the color control. These values are not controlled by the color control tile, so are not useful.

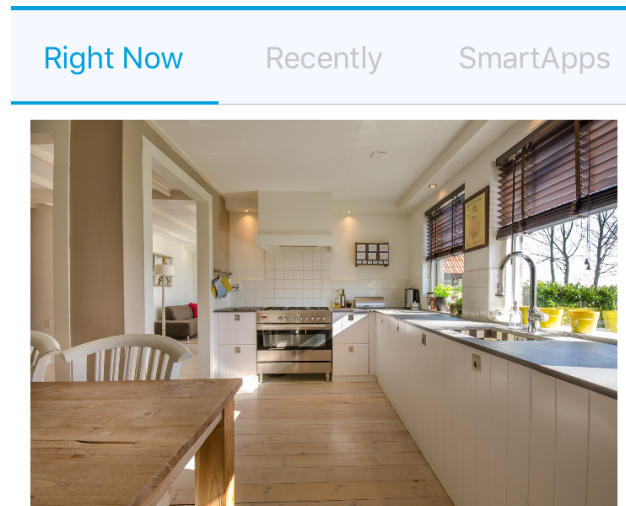
Tip: Check out the *Examples* (page 497) to see it in action!

133.4.5 Carousel Tile

A Carousel Tile is often used in conjunction with the `imageCapture` capability, to allow users to scroll through recent pictures.

Many of the camera Device Handlers will make use of the `carouselTile()`.

```
carouselTile("cameraDetails", "device.image", width: 3, height: 2) { }
```

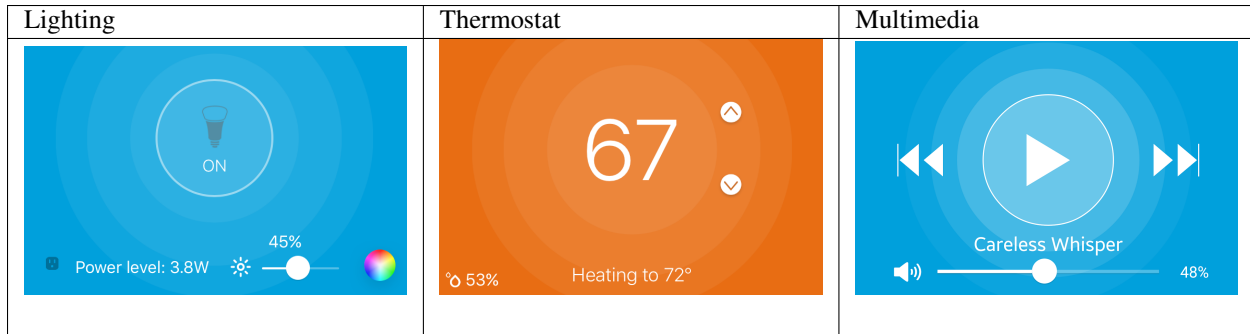


The Carousel Tile displays the ten most recent images captured within the past seven days.

Note: See *Capturing and Displaying Camera Pictures* (page 575) for more information on working with camera devices.

133.5 Multi-Attribute Tiles

Multi-Attribute Tiles combine multiple attributes into a single tile presented with a rich UI. Here are some of the types of tiles that you can create:



133.5.1 Basics

Multi-Attribute Tiles must be given a width of 6 and a height of 4. To enable this, the tiles builder of your Device Handler must use the new *6 X Unlimited* grid layout by specifying `scale: 2`:

```
tiles (scale: 2) {
    multiAttributeTile(name:"switch", type: "lighting", width: 6, height: 4, canChangeIcon: true) {
        ...
    }
}
```

133.5.2 Multi-Attribute Tile types

Multi-Attribute Tiles specify a type:

```
multiAttributeTile(name:"switch", type: "lighting", width: 6, height: 4) { ... }
```

The following types are supported, and each type is documented in detail below:

- "lighting"
- "thermostat"
- "mediaPlayer"
- "generic"

133.5.3 Attribute state and control keys

Like Single-Attribute Tiles, Multi-Attribute Tiles are associated with device attributes. As the name suggests, Multi-Attribute Tiles can be associated with more than one attribute, using `tileAttribute()` and `attributeState()`:

```
multiAttributeTile(name:"switch", type: "lighting", width: 6, height: 4, canChangeIcon: true) {
    tileAttribute ("device.switch", key: "PRIMARY_CONTROL") {
        attributeState "on", label:'${name}', action:"switch.off", icon:"st.lights.philips.hue-single
        attributeState "off", label:'${name}', action:"switch.on", icon:"st.lights.philips.hue-single
        attributeState "turningOn", label:'${name}', action:"switch.off", icon:"st.lights.philips.hue
        attributeState "turningOff", label:'${name}', action:"switch.on", icon:"st.lights.philips.hue
    }
    tileAttribute ("device.power", key: "SECONDARY_CONTROL") {
        attributeState "power", label:'Power level: ${currentValue}W', icon: "st.Appliances.appliance
    }
    tileAttribute ("device.level", key: "SLIDER_CONTROL") {
```

```

        attributeState "level", action:"switch level.setLevel"
    }
    tileAttribute ("device.color", key: "COLOR_CONTROL") {
        attributeState "color", action:"setAdjustedColor"
    }
}

```

The key difference between the Multi-Attribute Tile `tileAttribute` and the single-attribute `state` is the `key` option for `attributeState`. The `key` informs the platform the type of control for the tile attribute, which is then used to render the appropriate control. The keys commonly used for each type of tile will be discussed below, and a complete reference list is *also available* (page 494).

Every Multi-Attribute Tile must specify a `PRIMARY_CONTROL`. This is the main control, and will control the background color for the entire Multi-Attribute Tile (except for the *Thermostat Multi-Attribute Tile* (page 489)).

133.5.4 Lighting Multi-Attribute Tile

The lighting Multi-Attribute Tile makes it easy to create rich tiles for lighting devices. There are several ways a lighting Multi-Attribute Tile can be configured, depending on the type of bulb and its supported capabilities.

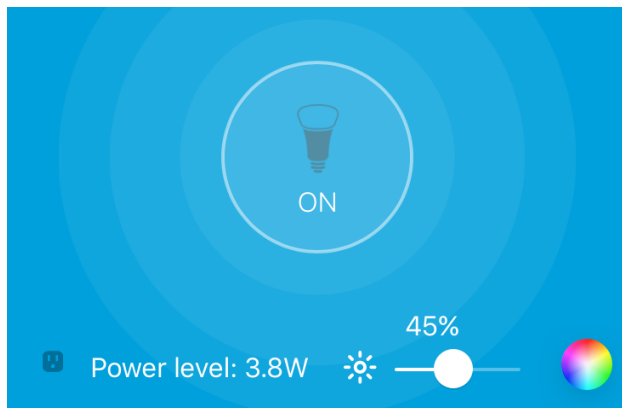
Consider the following Multi-Attribute Tile for a bulb that supports the `switch`, `colorControl`, `powerMeter`, and `switch-Level` capabilities:

```

multiAttributeTile(name:"switch", type: "lighting", width: 6, height: 4, canChangeIcon: true) {
    tileAttribute ("device.switch", key: "PRIMARY_CONTROL") {
        attributeState "on", label:'${name}', action:"switch.off", icon:"st.lights.philips.hue-single"
        attributeState "off", label:'${name}', action:"switch.on", icon:"st.lights.philips.hue-single"
        attributeState "turningOn", label:'${name}', action:"switch.off", icon:"st.lights.philips.hue"
        attributeState "turningOff", label:'${name}', action:"switch.on", icon:"st.lights.philips.hue"
    }
    tileAttribute ("device.power", key: "SECONDARY_CONTROL") {
        attributeState "power", label:'Power level: ${currentValue}W', icon: "st.Appliances.appliance"
    }
    tileAttribute ("device.level", key: "SLIDER_CONTROL") {
        attributeState "level", action:"switch level.setLevel"
    }
    tileAttribute ("device.color", key: "COLOR_CONTROL") {
        attributeState "color", action:"setColor"
    }
}

```

This tile renders as:



Note: Android will display the `SECONDARY_CONTROL` and `SLIDER_CONTROL` attribute values as a marquee when used in conjunction with `COLOR_CONTROL`.

The `tileAttribute` keys and their description used for the lighting Multi-Attribute Tile are summarized in the following table:

Key	Description
<code>PRIMARY_CONTROL</code>	Displays the status of the switch, and allows the switch state to be toggled when pressed.
<code>SECONDARY_CONTROL</code>	Used to display textual information. Often used to display power usage.
<code>SLIDER_CONTROL</code>	For bulbs that support the <code>switchLevel</code> capability, allows the user to set the switch level.
<code>COLOR_CONTROL</code>	For bulbs that support the <code>colorControl</code> capability, allows the user to select a color.

The command method specified by `action` will be called with a map that looks like this:

```
[red:241, hex:#F1E3FF, saturation:10.980392, blue:255, green:227, hue:75.0]
```

The values are summarized in the table below:

Key	Description
<code>red</code>	The red value chosen in the standard RGB color space
<code>hex</code>	The hexadecimal representation of the color chosen
<code>saturation</code>	The saturation value of the value chosen, between 0 and 100
<code>blue</code>	The blue value chosen in the standard RGB color space
<code>green</code>	The green value chosen in the standard RGB color space
<code>hue</code>	The hue value of the color chosen, between 0 and 100

You may also see a `level` and `alpha` attribute returned from the color control. These values are not controlled by the color palette, so are not useful.

Note: You may see code for Color Control bulbs that adjusts the Hue using some magic numbers and fun math.

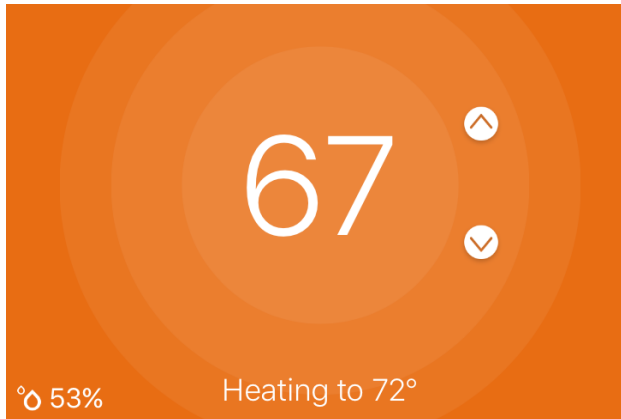
This is an artifact of the original Hue bulb sacrificing the ability to render greens in favor of more pleasant whites. This tradeoff threw off the actual colors version the apparent color on the color wheel. These calculations compensated for this behavior somewhat so that when you selected blue on the color wheel you actually saw blue on the bulb.

These adjustments would not apply to other color bulbs.

Tip: Check out the [Examples](#) (page 497) to see it in action!

133.5.5 Thermostat Multi-Attribute Tile

The Thermostat Multi-Attribute Tile allows for rich viewing and control of thermostat devices. Here's an image of a thermostat tile (when heating):



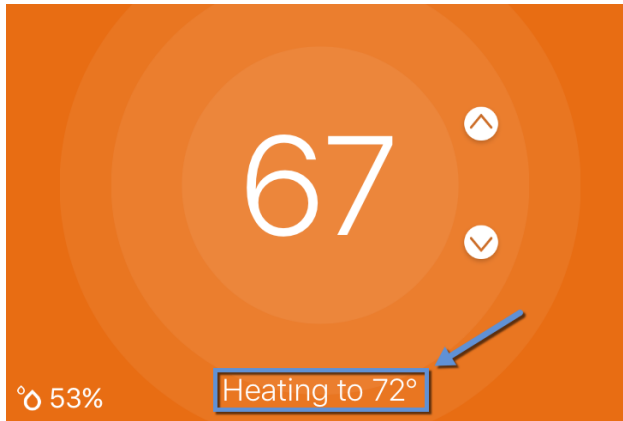
The tiles configuration for the above tile is:

```
multiAttributeTile(name:"thermostatFull", type:"thermostat", width:6, height:4) {
  tileAttribute("device.temperature", key: "PRIMARY_CONTROL") {
    attributeState("temp", label:'${currentValue}', unit:"dF", defaultState: true)
  }
  tileAttribute("device.temperature", key: "VALUE_CONTROL") {
    attributeState("VALUE_UP", action: "tempUp")
    attributeState("VALUE_DOWN", action: "tempDown")
  }
  tileAttribute("device.humidity", key: "SECONDARY_CONTROL") {
    attributeState("humidity", label:'${currentValue}%', unit:"%", defaultState: true)
  }
  tileAttribute("device.thermostatOperatingState", key: "OPERATING_STATE") {
    attributeState("idle", backgroundColor:"#00A0DC")
    attributeState("heating", backgroundColor:"#e86d13")
    attributeState("cooling", backgroundColor:"#00A0DC")
  }
  tileAttribute("device.thermostatMode", key: "THERMOSTAT_MODE") {
    attributeState("off", label:'${name}')
    attributeState("heat", label:'${name}')
    attributeState("cool", label:'${name}')
    attributeState("auto", label:'${name}')
  }
  tileAttribute("device.heatingSetpoint", key: "HEATING_SETPOINT") {
    attributeState("heatingSetpoint", label:'${currentValue}', unit:"dF", defaultState: true)
  }
  tileAttribute("device.coolingSetpoint", key: "COOLING_SETPOINT") {
    attributeState("coolingSetpoint", label:'${currentValue}', unit:"dF", defaultState: true)
  }
}
```

The below table summarizes the basic controls for a Thermostat Multi-Attribute Tile:

Key	Description
PRIMARY_CONTROL	Used to display the current temperature.
VALUE_CONTROL	Renders controls for increasing or decreasing the temperature.
SECONDARY_CONTROL	Used to display textual data about the thermostat, like humidity. Appears on the bottom-left of the tile.

In addition to the controls above, there are four additional controls that work together to show the status label at the bottom of the tile:



This label provides users with more information on the state of the thermostat. Additionally, thermostat tiles also look to the `OPERATING_STATE` attribute for its background color, falling back on the colors for `PRIMARY_CONTROL`.

In order to provide the relevant data to present the label, there are four additional attributes you should include:

Value	Description	Notes
<code>OPERATING_STATE</code>	What the thermostat is doing	The label will not show if <code>OPERATING_STATE</code> is omitted, as this is the baseline amount of meaningful information
<code>THERMOSTAT_MODE</code>	Thermostat Mode (i.e. Heat, Cool, or Auto)	This allows the user to know the Mode (and temperature) if the system is idle (e.g. “Idle—Heat at 66°”)
<code>HEATING_SETPOINT</code>	At which point the system will begin heating	Informs the user when heating will start (or stop, if currently heating)
<code>COOLING_SETPOINT</code>	At which point the system will begin cooling	Informs the user when cooling will start (or stop, if currently cooling)

Note: Only `OPERATING_STATE` is required to present the status label, but providing all four attributes will ensure the best experience for your users.

Tip: Check out the *Examples* (page 497) to see it in action!

133.5.6 Multimedia Multi-Attribute Tile

The Multimedia Multi-Attribute Tile is intended for devices that support the `musicPlayer` capability. It can render controls for playing, pausing, next/previous tracks, and volume levels for a music player. It can also display information about the currently playing track.



The code for this tiles configuration is shown below:

```
tiles (scale: 2) {
  multiAttributeTile(name: "mediaMulti", type:"mediaPlayer", width:6, height:4) {
    tileAttribute("device.status", key: "PRIMARY_CONTROL") {
```

```

        attributeState ("paused", label: "Paused",)
        attributeState ("playing", label: "Playing")
        attributeState ("stopped", label: "Stopped")
    }
    tileAttribute("device.status", key: "MEDIA_STATUS") {
        attributeState ("paused", label: "Paused", action: "music Player.play", nextState: "playing")
        attributeState ("playing", label: "Playing", action: "music Player.pause", nextState: "paused")
        attributeState ("stopped", label: "Stopped", action: "music Player.play", nextState: "playing")
    }
    tileAttribute("device.status", key: "PREVIOUS_TRACK") {
        attributeState ("status", action: "music Player.previousTrack", defaultState: true)
    }
    tileAttribute("device.status", key: "NEXT_TRACK") {
        attributeState ("status", action: "music Player.nextTrack", defaultState: true)
    }
    tileAttribute ("device.level", key: "SLIDER_CONTROL") {
        attributeState ("level", action: "music Player.setLevel")
    }
    tileAttribute ("device.mute", key: "MEDIA_MUTED") {
        attributeState ("unmuted", action: "music Player.mute", nextState: "muted")
        attributeState ("muted", action: "music Player.unmute", nextState: "unmuted")
    }
    tileAttribute("device.trackDescription", key: "MARQUEE") {
        attributeState ("trackDescription", label: "${currentValue}", defaultState: true)
    }
}

main "mediaMulti"
details(["mediaMulti"])
}

```

The `tileAttribute` control keys and their description used for the Multimedia Multi-Attribute Tile are summarized in the following table:

Key	Description
PRIMARY_CONTROL	Necessary to render the background of the tile
MEDIA_STATUS	Used to display and control the current play status (playing, paused, stopped)
PREVIOUS_TRACK	Renders a control for going to the previous track
NEXT_TRACK	Renders a control for going to the next track
SLIDER_CONTROL	Renders a control to select a volume level
MEDIA_MUTED	Allows the user to press the volume icon to mute
MARQUEE	Will display the currently playing track description below the PRIMARY_CONTROL. Use newlines (" <code>\n</code> ") to delimit fields such as title, artist, album, etc.

Note: The background color of the media Multi-Attribute Tile defaults to blue, and cannot be overridden.

Tip: Check out the *Examples* (page 497) to see it in action!

133.5.7 Generic Multi-Attribute Tile

If none of the predefined Multi-Attribute Tile types fit your needs, you can use the Generic Multi-Attribute Tile. The supported tile attribute types for the Generic Multi-Attribute Tile are shown in the following table:

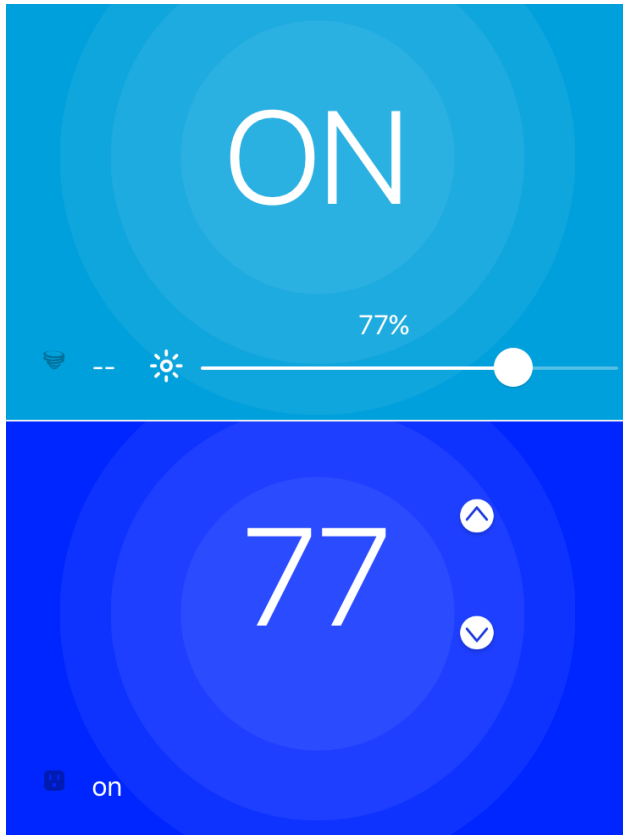
Key	Description
PRIMARY_CONTROL	The primary control tile for this device, controls the background color
SECONDARY_CONTROL	Displays textual data below the primary control
VALUE_CONTROL	Renders Up and Down buttons for increasing or decreasing values
SLIDER_CONTROL	Renders a slider control for selecting a value along a range
COLOR_CONTROL	Renders the color picker that allows users to select a color (useful for Color Control devices)

Here's an example of a generic tile:

```
multiAttributeTile(name:"sliderTile", type:"generic", width:6, height:4) {
  tileAttribute("device.switch", key: "PRIMARY_CONTROL") {
    attributeState "on", label:'${name}', backgroundColor:"#00A0DC", nextState:"turningOff"
    attributeState "off", label:'${name}', backgroundColor:"#ffffff", nextState:"turningOn"
    attributeState "turningOn", label:'${name}', backgroundColor:"#79b821", nextState:"turningOff"
    attributeState "turningOff", label:'${name}', backgroundColor:"#ffffff", nextState:"turningOn"
  }
  tileAttribute("device.level", key: "SECONDARY_CONTROL") {
    attributeState "level", icon: 'st.Weather.weather1', action:"randomizeLevel", defaultState: true
  }
  tileAttribute("device.level", key: "SLIDER_CONTROL") {
    attributeState "level", action:"switch_level.setLevel", defaultState: true
  }
}

multiAttributeTile(name:"valueTile", type:"generic", width:6, height:4) {
  tileAttribute("device.level", key: "PRIMARY_CONTROL") {
    attributeState "level", label:'${currentValue}', defaultState: true, backgroundColors:[
      [value: 0, color: "#ff0000"],
      [value: 20, color: "#ffff00"],
      [value: 40, color: "#00ff00"],
      [value: 60, color: "#00ffff"],
      [value: 80, color: "#0000ff"],
      [value: 100, color: "#ff00ff"]
    ]
  }
  tileAttribute("device.switch", key: "SECONDARY_CONTROL") {
    attributeState "on", label:'${name}', action:"switch.off", icon:"st.switches.switch.on", backgroundColor:"#ffffff", nextState:"turningOn"
    attributeState "off", label:'${name}', action:"switch.on", backgroundColor:"#ffffff", nextState:"turningOff"
    attributeState "turningOn", label:'...', action:"switch.off", icon:"st.switches.switch.on", backgroundColor:"#ffffff", nextState:"turningOff"
    attributeState "turningOff", label:'...', action:"switch.on", backgroundColor:"#ffffff", nextState:"turningOn"
  }
  tileAttribute("device.level", key: "VALUE_CONTROL") {
    attributeState "VALUE_UP", action: "levelUp"
    attributeState "VALUE_DOWN", action: "levelDown"
  }
}
```

The above tiles render as:



Tip: Check out the *Examples* (page 497) to see it in action!

133.5.8 Controls summary

The table below summarizes all the available control types. Not all controls are supported for all tile types; see the tile-specific documentation for more information.

Key	Description
COLOR_CONTROL	Displays a color palette for the user to select a color from.
COOLING_SETPOINT	Used by the <i>Thermostat Multi-Attribute Tile</i> (page 489).
HEATING_SETPOINT	Used by the <i>Thermostat Multi-Attribute Tile</i> (page 489).
MARQUEE	Displays a rotating marquee message beneath the PRIMARY_CONTROL.
MEDIA_MUTED	Allows the user to press the volume icon to mute on a <i>Multimedia Multi-Attribute Tile</i> (page 491).
MEDIA_STATUS	Used to display and control the current play status (playing, paused, stopped) on a <i>Multimedia Multi-Attribute Tile</i> (page 491).
NEXT_TRACK	Renders a control for going to the next track on a <i>Multimedia Multi-Attribute Tile</i> (page 491).
OPERATING_STATUS	Used by the <i>Thermostat Multi-Attribute Tile</i> (page 489).
PREVIOUS_TRACK	Renders a control for going to the previous track on a <i>Multimedia Multi-Attribute Tile</i> (page 491).
PRIMARY_CONTROL	All tiles must define a PRIMARY_CONTROL. Controls the background color of tile (except for the <i>Thermostat Multi-Attribute Tile</i> (page 489)), and specifies the attribute to show on the Device list views.
SECONDARY_CONTROL	Used to display textual information below the PRIMARY_CONTROL.
SLIDER_CONTROL	Displays a slider input; typically useful for attributes like bulb level or volume.
THERMOSTAT_MODE	Used by the <i>Thermostat Multi-Attribute Tile</i> (page 489).
VALUE_CONTROL	Renders Up and Down controls for increasing and decreasing an attribute's value by 1.

133.6 Color standards

SmartThings has defined a set of common colors for use in device tiles. Follow these standards when developing device tiles to ensure consistency within the SmartThings mobile app.

133.6.1 Colors

The following table lists the standard colors, their hexadecimal code, and a description of when to use the color:

Color	Hex code	Description	Color example
Blue	#00a0dc	Represents “on”-like device states	
White	#ffffff	Represents “off”-like device states	
Orange	#e86d13	Represents device states that require the user’s attention	
Gray	#cccccc	Represents “inactive” or “offline” device states	

Transition states (e.g., “Turning on”) should use the color of the transitioned-to state (e.g., blue for “Turning on”).

In addition to the colors above, tiles that display temperatures follow these standards (see the *Background color* (page 481) documentation to understand how the colors are interpolated between values):

Temperature value (Fahrenheit)	Hex code	Color example
31	#153591	
44	#1e9cbb	
59	#90d2a7	
74	#44b621	
84	#f1d801	
95	#d04e00	
96	#bc2323	

Tip: If your Device Handler needs to accommodate Celsius temperature values, you can convert the values above to Celsius, and expand the background colors out to include the range of both Celsius and Fahrenheit values. You can see an example of this [here](#).

133.6.2 Examples

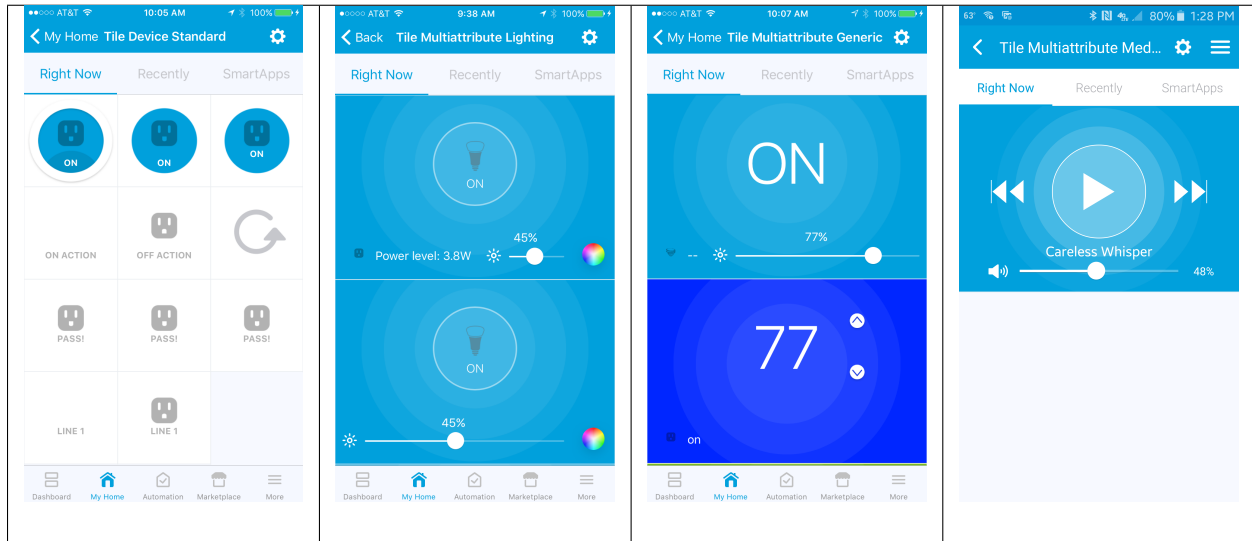
The following table contains several common device states and their tile color:

Attribute state	Color
Switch <i>on</i>	Blue-#00a0dc
Switch <i>off</i>	White-#ffffff
Motion <i>active</i>	Blue-#00a0dc
Motion <i>inactive</i>	White-#ffffff
Contact sensor <i>open</i>	Orange-##e86d13
Contact sensor <i>closed</i>	Blue-#00a0dc
Lock <i>locked</i>	Blue-#00a0dc
Lock <i>unlocked</i>	White-#ffffff
Presence <i>present</i>	Blue-#00a0dc
Presence <i>away</i>	Gray-#cccccc
Thermostat <i>cool</i>	Blue-#00a0dc
Thermostat <i>heat</i>	Orange-##e86d13
Siren <i>on</i>	Orange-##e86d13
Siren <i>off</i>	White-#ffffff
Water sensor <i>dry</i>	White-#ffffff
Water sensor <i>wet</i>	Blue-#00a0dc
Smoke detector <i>clear</i>	White-#ffffff
Smoke detector <i>detected</i>	Orange-#e86d13
Smoke detector <i>tested</i>	Orange-#e86d13

133.7 Additional information

- If using the `SECONDARY_CONTROL`, `SLIDER_CONTROL`, and `COLOR_CONTROL` controls in the same Multi-Attribute Tile, the values for the secondary and slider control will display as a Marquee on Android.
 - When specifying a Multi-Attribute Tile as the `main` tile, the `PRIMARY_CONTROL` tile attribute will display on the details list.
 - Tiles may not render the same across all mobile platforms. While we strive for a degree of consistency, it is still recommended to test your tiles on a variety of devices.
 - Remember that when tile definitions are consumed by the platform, the platform has no knowledge of device state, etc. Tiles are static in nature; keep this in mind as you design your Device Handler.
 - *6 x 1* tiles will actually render the tile that is used for the device in the Device List views. This is almost surely not what is desired, so it's recommended not to use *6 x 1* tiles.
-

133.8 Examples



We've created several Device Handlers for all the tiles discussed in this documentation. These are a great reference for seeing various tiles in action.

They are located in the `tiles-ux` package in the SmartThingsPublic GitHub Repository. Refer to the README in the package for information on installing and using the example devices.

Preferences

Device Handlers may specify simple preferences to allow the user to configure certain properties of their device.

134.1 Overview

When a user adds a device to SmartThings, they are given the option to name their device and select a room. If there are additional configuration options that you wish to expose to the user, you can specify them using `preferences`. They will appear on the same page as the device name preference.

134.2 Defining preferences

Device preferences should be placed in the Device Handler's metadata. They can appear anywhere in the metadata definition.

```
metadata {
  definition(...) {...}
  tiles() {...}
  preferences {
    input "tempOffset", "number", title: "Degrees", description: "Adjust temperature by this many",
        range: "...", displayDuringSetup: false
  }
}
```

134.3 Device preferences are flat

Device preferences are static, and single-page.

Multiple page preferences and dynamic preferences pages are not supported in Device Handlers. Device Handler preferences are a simple list of inputs:

```
preferences {
  input name: "text", type: "text", title: "Text", description: "Enter Text", required: true
}
```

134.4 Display on setup

Use `displayDuringSetup: true` to force the preference input to be displayed when the device is being added to SmartThings:

```
preferences {
    input name: "email", type: "email", title: "Email", description: "Enter Email Address", required: true,
           displayDuringSetup: true
}
```

Preferences that do not specify this value, or specify `displayDuringSetup: false`, will only appear when the user presses the Settings button on the Device in the mobile application.

134.5 Supported input types

The following input types are supported in Device Handler preferences:

- bool
 - decimal
 - email
 - enum
 - number
 - password
 - phone
 - time
 - text
-

134.6 Getting preference input values

Just as with SmartApp preferences, the name of the preferences input is a reference to the preference value:

```
metadata {
    definition(...) {...}
    tiles() {...}
    preferences {
        input "tempOffset", "number", title: "Degrees", description: "Adjust temperature by this many"
    }
}

def someCommandMethod() {
    if (tempOffset) {
        // handle offset value
    }
}
```

Note: Preference values are only available to the Device Handler when it is executing in response to Events or commands. It is not possible to use preference values in other metadata definitions, including `tiles()`.

134.7 Example

```
metadata {
    simulator {
        // TODO: define status and reply messages here
    }

    tiles {
        // TODO: define your main and details tiles here
    }

    preferences {
        input name: "email", type: "email", title: "Email", description: "Enter Email Address", required: true
        input name: "text", type: "text", title: "Text", description: "Enter Text", required: true
        input name: "number", type: "number", title: "Number", description: "Enter number", required: true
        input name: "bool", type: "bool", title: "Bool", description: "Enter boolean", required: true
        input name: "password", type: "password", title: "password", description: "Enter password", required: true
        input name: "phone", type: "phone", title: "phone", description: "Enter phone", required: true
        input name: "decimal", type: "decimal", title: "decimal", description: "Enter decimal", required: true
        input name: "time", type: "time", title: "time", description: "Enter time", required: true
        input name: "options", type: "enum", title: "enum", options: ["Option 1", "Option 2"], description: "Enter options"
    }
}

def someCommand() {
    log.debug "email: $email"
    log.debug "text: $text"
    log.debug "bool: $bool"
    log.debug "password: $password"
    log.debug "phone: $phone"
    log.debug "decimal: $decimal"
    log.debug "time: $time"
    log.debug "options: $options"
}
```

134.8 Additional notes

- Setting a default value (`defaultValue: "foobar"`) for an input may render that selection in the mobile app, but the user still needs to enter data in that field. It's recommended to not use `defaultValue` to avoid confusion.

Parse and Events

The `parse` method is the core method in a typical Device Handler.

135.1 Overview

All messages from the device are passed to the `parse()` method. It is responsible for turning those messages into something the SmartThings platform can understand.

Because the `parse()` method is responsible for handling raw device messages, their implementations vary greatly across different device types. This document will not discuss all these different scenarios (see the Z-Wave Device Handler Guide or ZigBee Device Handler guide for protocol-specific information).

Consider an example of a simplified `parse()` method (modified from the Centralite Switch):

```
def parse(String description) {
    log.debug "parse description: $description"

    def attrName = null
    def attrValue = null

    if (description?.startsWith("on/off:")) {
        log.debug "switch command"
        attrName = "switch"
        attrValue = description?.endsWith("1") ? "on" : "off"
    }

    def result = createEvent(name: attrName, value: attrValue)

    log.debug "Parse returned ${result?.descriptionText}"
    return result
}
```

Our `parse()` method inspects the passed-in description, and creates an Event with name “switch” and a value of “on” or “off”. It then returns the created Event, where the SmartThings platform will handle firing the Event and notifying any SmartApps subscribed to that Event.

135.2 Parse, Events, and Attributes

Recall that the “switch” capability specifies an attribute of “switch”, with possible values “on” and “off”. *The `parse()` method is responsible for creating events for the attributes of that device’s capabilities.*

That is a critical point to understand about Device Handlers - it is what allows SmartApps to respond to Event subscriptions!

Note: Only events that constitute a state change are propagated through the SmartThings platform. A state change is when a particular attribute of the device changes. This is handled automatically by the platform, but should you want to override that behavior, you can do so by specifying the `isStateChange` parameter discussed below.

135.2.1 Creating Events

Use the `createEvent()` method to create events in your Device Handler. It takes a map of parameters as an argument. You should provide the `name` and `value` at a minimum.

Important: The `createEvent` just creates a data structure (a Map) with information about the Event. *It does not actually fire an Event.*

Only by returning that created map from your `parse` method will an Event be fired by the SmartThings platform.

The parameters you can pass to `createEvent` are:

***name* (required)** String - The name of the Event. Typically corresponds to an attribute name of the device-handler’s capabilities.

***value* (required)** The value of the Event. The value is stored as a String, but you can pass in numbers or other objects. SmartApps will be responsible for parsing the Event’s value into back to its desired form (e.g., parsing a number from a string)

descriptionText String - The description of this Event. This appears in the mobile application activity feed for the device. If not specified, this will be created using the Event name and value.

displayed boolean - `true` to display this Event in the mobile application activity feed. `false` to not display this Event. Defaults to `true`.

linkText String - Name of the Event to show in the mobile application activity feed, if specified.

isStateChange boolean - `true` if this Event caused the device’s attribute to change state. `false` otherwise. If not provided, `createEvent` will populate this based on the current state of the device.

unit String - a unit string, if desired. This will be used to create the `descriptionText` if it (the `descriptionText` parameter) is not specified.

135.2.2 Multiple Events

You are not limited to returning a single Event map from your `parse` method.

You can return a list of Event maps to tell the SmartThings platform to generate multiple events:

```
def parse(String description) {  
    ...  
}
```

```
def evt1 = createEvent(name: "someName", value: "someValue")
def evt2 = createEvent(name: "someOtherName", value: "someOtherValue")

return [evt1, evt2]
}
```

135.2.3 Generating Events outside of parse

If you need to generate an Event outside of the `parse()` method, you can use the `sendEvent()` method. It simply calls `createEvent()` *and* fires the Event. You pass in the same parameters as you do to `createEvent()`.

135.3 Tips

When creating a Device Handler, determining what messages need to be handled by the `parse()` method varies by device. A common practice to figure out what messages need to be handled is to simply log the messages in your `parse()` method (`log.debug "description: $description"`). This allows you to see what the incoming message is for various actuations or states.

Z-Wave Primer

This document covers some important aspects of the Z-Wave application-level standard that you may come in contact with when developing Device Handlers for Z-Wave devices. If you are already familiar with Z-Wave development, you can learn how SmartThings integrates with it in [Building Z-Wave Device Handlers](#). You can also consult the [Z-Wave public specification](#) for more information about the Z-Wave protocol.

136.1 Command classes

Z-Wave device messages are all called “commands”, even if they are just info reports or other kinds of communications. They are organized into *command classes* which group related functionality together. Some devices list which command classes they support in their manuals.

There is a list of the command classes that SmartThings supports here: [Z-Wave Command Reference](#). Notice some of them have multiple versions. The Z-Wave standard occasionally adds a new version of a command class that may add new commands or add more data fields to existing commands. New versions are backwards-compatible and generally our command parsing system can handle different versions interchangeably, but you may need to specify a specific version in some cases.

Some commonly seen command classes:

- [0x20 Basic](#)
A generalized get/set/report command class that all devices support. It is usually mapped to another more specific command class, like Switch Binary for switches or Sensor Binary for sensors.
- [0x25 Switch Binary](#)
Control of on/off switches.
- [0x26 Switch Multilevel](#)
Control of dimmer switches.
- [0x30 Sensor Binary](#)
Sensors with two states, such as motion detectors and open/closed sensors.
- [0x31 Sensor Multilevel](#)
Sensors that report a numeric value, like temperature or illuminance.
- [0x32 Meter](#)
Outlets and meters that measure energy use.
- [0x71 Alarm/Notification](#)

The *Alarm* command class was renamed to *Notification* in version 3.

Used by sensors and other devices to report events.

- [0x70 Configuration](#)
See *Configuration* section below.
 - [0x80 Battery](#)
Battery level reporting for battery powered devices.
 - [0x84 Wake Up](#)
See *Listening and Sleepy Devices* section below.
 - [0x85 Association](#)
See *Association* section below.
 - [0x86 Version](#)
All devices report their Z-Wave framework and firmware version on request.
 - [0x72 Manufacturer Specific](#)
All devices report their manufacturer and model (via numeric code).
 - [0x98 Security](#)
Commands to and from security-sensitive devices can be sent encrypted by wrapping them in *SecurityMessageEncapsulation* commands.
 - [0x60 Multi-Channel/Multi-Instance](#)
The *Multi Instance* command class was renamed to *Multi Channel* in version 3. It is used by devices to distinguish between multiple control or reporting end points.
-

136.2 Listening and sleepy devices

Z-Wave devices that are plugged in to power are called **listening** devices because they keep their receiver on all the time. Listening devices act as repeaters and therefore extend the Z-Wave mesh network.

Battery powered Z-Wave devices such as sensors or remote controllers are **sleepy** – they turn off their receivers to save energy, so you can't send them commands at any time. Instead, they wake up at a regular interval and send a *WakeUpNotification* to alert other devices that they will be listening for incoming commands for the next few seconds. The *WakeUpIntervalSet* command is used to configure both how often the device will wake up and which controller it will send its *WakeUpNotification* to. When the controller gets the *WakeUpNotification* and has no commands to send to the device, it can send *WakeUpNoMoreInformation* to tell the device that it can go back to sleep.

Some battery powered devices like door locks and thermostats have to be able to receive commands at any time. These are known as **beamable** devices, because they wake up for only a tiny slice of time each second or quarter-second and listen for a “beam”. Thus, the sending device must “beam” the receiving device for a full second to wake it up fully before sending a command. This makes communication with these devices take a significantly longer time than with a normal listening device.

136.3 Configuration

A Z-Wave device can use the *Configuration* command class to allow the user to change its settings. Configuration parameters and their interpretation vary between device models, and are usually detailed in the device's manual or technical documentation.

The command class includes commands to read and set configuration parameter values. One thing to be careful of is that the `ConfigurationSet` command encodes the setting value in a 1, 2, or 4 byte format, and many devices will only properly interpret the value if it is sent in the same byte format. When sending a `ConfigurationSet`, make sure to set the 'size' argument to the same value as it has in an incoming `ConfigurationReport` from the device for the parameter number in question.

136.4 Association

The `Association` command class is used to tell a Z-Wave device that it should send updates to another device. It provides the ability to add associated devices to different numbered groups that can have different meanings. This functionality is used in a few different ways, often detailed in the device's manual or technical documentation:

- Some sensors will send reports of the events they detect only to devices that have been added to a specific association group.
- Many sensors will send `BasicSet` commands to associated devices, for example to turn a light on when a door opens and off when it closes.
- Some devices have multiple groups for different uses, like group 1 gets sent `BasicSet` commands, group 2 gets sent `SensorBinaryReport` events, and group 3 gets sent `BatteryReport` updates.
- Most door locks will send status updates to associated devices when they are locked or unlocked manually.

The SmartThings Hub automatically adds itself to association group 1 when a device that supports association joins the network. If this is inappropriate for your Device Handler, your Device Handler can use `AssociationRemove` to undo it. To associate to a group higher than 1, the Device Handler can send `AssociationSet`. The Hub's node ID is provided to Device Handler code in the variable `zwaveHubNodeId`.

Building Z-Wave Device Handlers

The Z-Wave public specification is available [here](#). SmartThings provides custom Z-Wave command objects that represent the standard commands and messages that Z-Wave devices use to send and request information.

137.1 Parsing Events

When Events from Z-Wave devices are passed into your Device Handler's parse method, they are in an encoded string format. The first thing your parse method should do is call `zwave.parse` on the description string to convert it to a Z-Wave command object. The object's class is one of the subclasses of `physicalgraph.zwave.Command` that can be found in the [Z-Wave Command Reference](#). If the description string does not represent a valid Z-Wave command, `zwave.parse` will return null.

```
def parse(String description) {
    def result = null
    def cmd = zwave.parse(description)
    if (cmd) {
        result = zwaveEvent(cmd)
        log.debug "Parsed ${cmd} to ${result.inspect()}"
    } else {
        log.debug "Non-parsed event: ${description}"
    }
    return result
}
```

Once you have a command object, the recommended way of handling it is to pass it to a overloaded function such as `zwaveEvent()` used in this example, with different argument types for the different types of commands you intend to handle:

```
def zwaveEvent(physicalgraph.zwave.commands.basicv1.BasicReport cmd)
{
    def result
    if (cmd.value == 0) {
        result = createEvent(name: "switch", value: "off")
    } else {
        result = createEvent(name: "switch", value: "on")
    }
    return result
}

def zwaveEvent(physicalgraph.zwave.commands.meterv3.MeterReport cmd) {
```

```
def result
  if (cmd.scale == 0) {
    result = createEvent(name: "energy", value: cmd.scaledMeterValue, unit: "kWh")
  } else if (cmd.scale == 1) {
    result = createEvent(name: "energy", value: cmd.scaledMeterValue, unit: "kVAh")
  } else {
    result = createEvent(name: "power", value: cmd.scaledMeterValue, unit: "W")
  }
  return result
}

def zwaveEvent(physicalgraph.zwave.Command cmd) {
  // This will capture any commands not handled by other instances of zwaveEvent
  // and is recommended for development so you can see every command the device sends
  return createEvent(descriptionText: "${device.displayName}: ${cmd}")
}
```

Remember that when you use `createEvent()` to build an Event, the resulting map must be returned from `parse()` for the Event to be sent. For information about `createEvent`, see the [Creating Events](#) section.

As the [Z-Wave Command Reference](#) shows, many Z-Wave command classes have multiple versions. By default, `zwave.parse()` will parse a command using the highest version of the command class. If the device is sending an earlier version of the command, some fields may be missing, or the command may fail to parse and return `null`. To fix this, you can pass in a map as the second argument to `zwave.parse()` to tell it which version of each command class to use:

```
zwave.parse(description, [0x26: 1, 0x70: 1])
```

This example will use version 1 of `SwitchMultilevel (0x26)` and `Configuration (0x70)` instead of the highest versions.

137.2 Sending commands

To send a Z-Wave command to the device, you must create the command object, call `format()` on it to convert it to the encoded string representation, and return it from the command method.

```
def on() {
  return zwave.basicV1.basicSet(value: 0xFF).format()
}
```

There is a shorthand provided to create command objects: `zwave.basicV1.basicSet(value: 0xFF)` is the same as `new physicalgraph.zwave.commands.basicv1.BasicSet(value: 0xFF)`. Note the different capitalization of the command name and the ‘V’ in the command class name.

The value `0xFF` passed in to the command is a hexadecimal number. Many Z-Wave commands use 8-bit integers to represent device state. Generally 0 means “off” or “inactive”, 1-99 are used as percentage values for a variable level attribute, and `0xFF` or 255 (the highest value) means “on” or “detected”.

If you want to send more than one Z-Wave command, you can return a list of formatted command strings. It is often a good idea to add a delay between commands to give the device an opportunity to finish processing each command and possibly send a response before receiving the next command. To add a delay between commands, include a string of the form `"delay N"` where `N` is the number of milliseconds to delay. There is a helper method `delayBetween()` that will take a list of commands and insert delay commands between them:

```
def off() {
  delayBetween([
    zwave.basicV1.basicSet(value: 0).format(),
    zwave.switchBinaryV1.switchBinaryGet().format()
  ], 100)
}
```

This example returns the output of `delayBetween`, and thus will send a `BasicSet` command, followed by a 100 ms delay (0.1 seconds), then a `SwitchBinaryGet` command in order to check immediately that the state of the switch was indeed changed by the `set` command.

137.3 Sending commands in response to Events

In some situations, instead of sending a command in response to a request by the user, you want to automatically send a command to the device on receipt of a Z-Wave command.

If you return a list from the parse method, each item of the list will be evaluated separately. Items that are maps will be processed as Events as usual and sent to subscribed SmartApps and mobile clients. Returned items that are `HubAction` items, however, will be sent via the Hub to the device, in much the same way as formatted commands returned from command methods. The easiest way to send a command to a device in response to an Event is the `response()` helper, which takes a Z-Wave command or encoded string and supplies a `HubAction`:

```
def zwaveEvent(physicalgraph.zwave.commands.wakeupv1.WakeUpNotification cmd)
{
  def event = createEvent(descriptionText: "${device.displayName} woke up", displayed: false)
  def cmds = []
  cmds << zwave.batteryV1.batteryGet().format()
  cmds << "delay 1200"
  cmds << zwave.wakeupV1.wakeUpNoMoreInformation().format()
  [event, response(cmds)] // return a list containing the event and the result of response()
}
```

The above example uses the `response()` helper to send Z-Wave commands and delay commands to the device whenever a `WakeUpNotification` Event is received. The reception of this Event that indicates that the sleepy device is temporarily listening for commands. In addition to creating a hidden Event, the handler will send a `BatteryGet` request, wait 1.2 seconds for a response, and then issue a `WakeUpNoMoreInformation` command to tell the device it can go back to sleep to save battery.

Z-Wave Example

Below is a Device Handler code sample with examples of many common commands and parsed events.

You can also view this example in [GitHub here](#).

```

metadata {
    definition (name: "Z-Wave Device Reference", author: "SmartThings") {
        capability "Actuator"
        capability "Switch"
        capability "Polling"
        capability "Refresh"
        capability "Temperature Measurement"
        capability "Sensor"
        capability "Battery"
    }
}

simulator {
    // These show up in the IDE simulator "messages" drop-down to test
    // sending event messages to your device handler
    status "basic report on":
        zwave.basicV1.basicReport (value:0xFF).incomingMessage ()
    status "basic report off":
        zwave.basicV1.basicReport (value:0).incomingMessage ()
    status "dimmer switch on at 70%":
        zwave.switchMultilevelV1.switchMultilevelReport (value:70).incomingMessage ()
    status "basic set on":
        zwave.basicV1.basicSet (value:0xFF).incomingMessage ()
    status "temperature report 70°F":
        zwave.sensorMultilevelV2.sensorMultilevelReport (scaledSensorValue: 70)
    status "low battery alert":
        zwave.batteryV1.batteryReport (batteryLevel:0xFF).incomingMessage ()
    status "multichannel sensor":
        zwave.multiChannelV3.multiChannelCmdEncap (sourceEndPoint:1, destinationEndPoint:2)

    // simulate turn on
    reply "2001FF,delay 5000,2002": "command: 2503, payload: FF"

    // simulate turn off
    reply "200100,delay 5000,2002": "command: 2503, payload: 00"
}

tiles {
    standardTile("switch", "device.switch", width: 2, height: 2,
        canChangeIcon: true) {

```

```

        state "on", label: '${name}', action: "switch.off",
            icon: "st.unknown.zwave.device", backgroundColor: "#79b821"
        state "off", label: '${name}', action: "switch.on",
            icon: "st.unknown.zwave.device", backgroundColor: "#ffffff"
    }
    standardTile("refresh", "command.refresh", inactiveLabel: false,
        decoration: "flat") {
        state "default", label: '', action: "refresh.refresh",
            icon: "st.secondary.refresh"
    }

    valueTile("battery", "device.battery", inactiveLabel: false,
        decoration: "flat") {
        state "battery", label: '${currentValue}% battery', unit: ""
    }

    valueTile("temperature", "device.temperature") {
        state("temperature", label: '${currentValue}°',
            backgroundColors: [
                [value: 31, color: "#153591"],
                [value: 44, color: "#1e9cbb"],
                [value: 59, color: "#90d2a7"],
                [value: 74, color: "#44b621"],
                [value: 84, color: "#f1d801"],
                [value: 95, color: "#d04e00"],
                [value: 96, color: "#bc2323"]
            ])
    }
}

main(["switch", "temperature"])
details(["switch", "temperature", "refresh", "battery"])
}

def parse(String description) {
    def result = null
    def cmd = zwave.parse(description, [0x60: 3])
    if (cmd) {
        result = zwaveEvent(cmd)
        log.debug "Parsed ${cmd} to ${result.inspect()}"
    } else {
        log.debug "Non-parsed event: ${description}"
    }
    result
}

def zwaveEvent(physicalgraph.zwave.commands.basicv1.BasicReport cmd)
{
    def result = []
    result << createEvent(name: "switch", value: cmd.value ? "on" : "off")

    // For a multilevel switch, cmd.value can be from 1-99 to represent
    // dimming levels
    result << createEvent(name: "level", value: cmd.value, unit: "%",
        descriptionText: "${device.displayName} dimmed ${cmd.value==255 ? 100 : cmd.value}%"
    )
    result
}

```



```

}

def zwaveEvent(physicalgraph.zwave.commands.switchbinaryv1.SwitchBinaryReport cmd) {
    createEvent(name:"switch", value: cmd.value ? "on" : "off")
}

def zwaveEvent(physicalgraph.zwave.commands.switchmultilevelv3.SwitchMultilevelReport cmd) {
    def result = []
    result << createEvent(name:"switch", value: cmd.value ? "on" : "off")
    result << createEvent(name:"level", value: cmd.value, unit:"%",
        descriptionText:"${device.displayName} dimmed ${cmd.value==255 ? 100 : cmd.value}")
    result
}

def zwaveEvent(physicalgraph.zwave.commands.meterv1.MeterReport cmd) {
    def result
    if (cmd.scale == 0) {
        result = createEvent(name: "energy", value: cmd.scaledMeterValue,
            unit: "kWh")
    } else if (cmd.scale == 1) {
        result = createEvent(name: "energy", value: cmd.scaledMeterValue,
            unit: "kVAh")
    } else {
        result = createEvent(name: "power",
            value: Math.round(cmd.scaledMeterValue), unit: "W")
    }
    result
}

def zwaveEvent(physicalgraph.zwave.commands.meterv3.MeterReport cmd) {
    def map = null
    if (cmd.meterType == 1) {
        if (cmd.scale == 0) {
            map = [name: "energy", value: cmd.scaledMeterValue,
                unit: "kWh"]
        } else if (cmd.scale == 1) {
            map = [name: "energy", value: cmd.scaledMeterValue,
                unit: "kVAh"]
        } else if (cmd.scale == 2) {
            map = [name: "power", value: cmd.scaledMeterValue, unit: "W"]
        } else {
            map = [name: "electric", value: cmd.scaledMeterValue]
            map.unit = ["pulses", "V", "A", "R/Z", ""][cmd.scale - 3]
        }
    } else if (cmd.meterType == 2) {
        map = [name: "gas", value: cmd.scaledMeterValue]
        map.unit = ["m^3", "ft^3", "", "pulses", ""][cmd.scale]
    } else if (cmd.meterType == 3) {
        map = [name: "water", value: cmd.scaledMeterValue]
        map.unit = ["m^3", "ft^3", "gal"][cmd.scale]
    }
    if (map) {
        if (cmd.previousMeterValue && cmd.previousMeterValue != cmd.meterValue) {
            map.descriptionText = "${device.displayName} ${map.name} is ${map.value} ${map.unit}"
        }
        createEvent(map)
    } else {

```

```
        null
    }
}

def zwaveEvent(physicalgraph.zwave.commands.sensorbinaryv2.SensorBinaryReport cmd) {
    def result
    switch (cmd.sensorType) {
        case 2:
            result = createEvent(name:"smoke",
                                value: cmd.sensorValue ? "detected" : "closed")
            break
        case 3:
            result = createEvent(name:"carbonMonoxide",
                                value: cmd.sensorValue ? "detected" : "clear")
            break
        case 4:
            result = createEvent(name:"carbonDioxide",
                                value: cmd.sensorValue ? "detected" : "clear")
            break
        case 5:
            result = createEvent(name:"temperature",
                                value: cmd.sensorValue ? "overheated" : "normal")
            break
        case 6:
            result = createEvent(name:"water",
                                value: cmd.sensorValue ? "wet" : "dry")
            break
        case 7:
            result = createEvent(name:"temperature",
                                value: cmd.sensorValue ? "freezing" : "normal")
            break
        case 8:
            result = createEvent(name:"tamper",
                                value: cmd.sensorValue ? "detected" : "okay")
            break
        case 9:
            result = createEvent(name:"aux",
                                value: cmd.sensorValue ? "active" : "inactive")
            break
        case 0x0A:
            result = createEvent(name:"contact",
                                value: cmd.sensorValue ? "open" : "closed")
            break
        case 0x0B:
            result = createEvent(name:"tilt", value: cmd.sensorValue ? "detected" : "okay")
            break
        case 0x0C:
            result = createEvent(name:"motion",
                                value: cmd.sensorValue ? "active" : "inactive")
            break
        case 0x0D:
            result = createEvent(name:"glassBreak",
                                value: cmd.sensorValue ? "detected" : "okay")
            break
        default:
            result = createEvent(name:"sensor",
                                value: cmd.sensorValue ? "active" : "inactive")
            break
    }
}
```

```
    }
    result
}

def zwaveEvent(physicalgraph.zwave.commands.sensorbinaryv1.SensorBinaryReport cmd)
{
    // Version 1 of SensorBinary doesn't have a sensor type
    createEvent(name:"sensor", value: cmd.sensorValue ? "active" : "inactive")
}

def zwaveEvent(physicalgraph.zwave.commands.sensormultilevelv5.SensorMultilevelReport cmd)
{
    def map = [ displayed: true, value: cmd.scaledSensorValue.toString() ]
    switch (cmd.sensorType) {
        case 1:
            map.name = "temperature"
            map.unit = cmd.scale == 1 ? "F" : "C"
            break;
        case 2:
            map.name = "value"
            map.unit = cmd.scale == 1 ? "%" : ""
            break;
        case 3:
            map.name = "illuminance"
            map.value = cmd.scaledSensorValue.toInteger().toString()
            map.unit = "lux"
            break;
        case 4:
            map.name = "power"
            map.unit = cmd.scale == 1 ? "Btu/h" : "W"
            break;
        case 5:
            map.name = "humidity"
            map.value = cmd.scaledSensorValue.toInteger().toString()
            map.unit = cmd.scale == 0 ? "%" : ""
            break;
        case 6:
            map.name = "velocity"
            map.unit = cmd.scale == 1 ? "mph" : "m/s"
            break;
        case 8:
        case 9:
            map.name = "pressure"
            map.unit = cmd.scale == 1 ? "inHg" : "kPa"
            break;
        case 0xE:
            map.name = "weight"
            map.unit = cmd.scale == 1 ? "lbs" : "kg"
            break;
        case 0xF:
            map.name = "voltage"
            map.unit = cmd.scale == 1 ? "mV" : "V"
            break;
        case 0x10:
            map.name = "current"
            map.unit = cmd.scale == 1 ? "mA" : "A"
            break;
        case 0x12:
    }
```

```

        map.name = "air flow"
        map.unit = cmd.scale == 1 ? "cfm" : "m^3/h"
        break;
    case 0x1E:
        map.name = "loudness"
        map.unit = cmd.scale == 1 ? "dBA" : "dB"
        break;
    }
    createEvent(map)
}

// Many sensors send BasicSet commands to associated devices.
// This is so you can associate them with a switch-type device
// and they can directly turn it on/off when the sensor is triggered.
def zwaveEvent(physicalgraph.zwave.commands.basicv1.BasicSet cmd)
{
    createEvent(name:"sensor", value: cmd.value ? "active" : "inactive")
}

def zwaveEvent(physicalgraph.zwave.commands.batteryv1.BatteryReport cmd) {
    def map = [ name: "battery", unit: "%" ]
    if (cmd.batteryLevel == 0xFF) { // Special value for low battery alert
        map.value = 1
        map.descriptionText = "${device.displayName} has a low battery"
        map.isStateChange = true
    } else {
        map.value = cmd.batteryLevel
    }
    // Store time of last battery update so we don't ask every wakeup, see WakeUpNotification handler
    state.lastbatt = new Date().time
    createEvent(map)
}

// Battery powered devices can be configured to periodically wake up and
// check in. They send this command and stay awake long enough to receive
// commands, or until they get a WakeUpNoMoreInformation command that
// instructs them that there are no more commands to receive and they can
// stop listening.
def zwaveEvent(physicalgraph.zwave.commands.wakeupv2.WakeUpNotification cmd)
{
    def result = [createEvent(descriptionText: "${device.displayName} woke up", isStateChange: false)]

    // Only ask for battery if we haven't had a BatteryReport in a while
    if (!state.lastbatt || (new Date().time) - state.lastbatt > 24*60*60*1000) {
        result << response(zwave.batteryV1.batteryGet())
        result << response("delay 1200") // leave time for device to respond to batteryGet
    }
    result << response(zwave.wakeupV1.wakeUpNoMoreInformation())
    result
}

def zwaveEvent(physicalgraph.zwave.commands.associationv2.AssociationReport cmd) {
    def result = []
    if (cmd.nodeId.any { it == zwaveHubNodeId }) {
        result << createEvent(descriptionText: "${device.displayName} is associated in group $cmd.groupingIdentifier")
    } else if (cmd.groupingIdentifier == 1) {
        // We're not associated properly to group 1, set association
        result << createEvent(descriptionText: "Associating ${device.displayName} in group $cmd.groupingIdentifier")
    }
    result
}

```

```

        result << response(zwave.associationV1.associationSet(groupingIdentifier: cmd.grouping
    }
    result
}

// Devices that support the Security command class can send messages in an
// encrypted form; they arrive wrapped in a SecurityMessageEncapsulation
// command and must be unencapsulated
def zwaveEvent(physicalgraph.zwave.commands.securityv1.SecurityMessageEncapsulation cmd) {
    def encapsulatedCommand = cmd.encapsulatedCommand([0x98: 1, 0x20: 1])

    // can specify command class versions here like in zwave.parse
    if (encapsulatedCommand) {
        return zwaveEvent(encapsulatedCommand)
    }
}

// MultiChannelCmdEncap and MultiInstanceCmdEncap are ways that devices
// can indicate that a message is coming from one of multiple subdevices
// or "endpoints" that would otherwise be indistinguishable
def zwaveEvent(physicalgraph.zwave.commands.multichannelv3.MultiChannelCmdEncap cmd) {
    def encapsulatedCommand = cmd.encapsulatedCommand([0x30: 1, 0x31: 1])

    // can specify command class versions here like in zwave.parse
    log.debug ("Command from endpoint ${cmd.sourceEndPoint}: ${encapsulatedCommand}")

    if (encapsulatedCommand) {
        return zwaveEvent(encapsulatedCommand)
    }
}

def zwaveEvent(physicalgraph.zwave.commands.multichannelv3.MultiInstanceCmdEncap cmd) {
    def encapsulatedCommand = cmd.encapsulatedCommand([0x30: 1, 0x31: 1])

    // can specify command class versions here like in zwave.parse
    log.debug ("Command from instance ${cmd.instance}: ${encapsulatedCommand}")

    if (encapsulatedCommand) {
        return zwaveEvent(encapsulatedCommand)
    }
}

def zwaveEvent(physicalgraph.zwave.Command cmd) {
    createEvent(descriptionText: "${device.displayName}: ${cmd}")
}

def on() {
    delayBetween([
        zwave.basicV1.basicSet(value: 0xFF).format(),
        zwave.basicV1.basicGet().format()
    ], 5000) // 5 second delay for dimmers that change gradually, can be left out for immediate
}

def off() {
    delayBetween([
        zwave.basicV1.basicSet(value: 0x00).format(),
        zwave.basicV1.basicGet().format()
    ], 5000) // 5 second delay for dimmers that change gradually, can be left out for immediate
}

```

```
}  
  
def refresh() {  
    // Some examples of Get commands  
    delayBetween([  
        zwave.switchBinaryV1.switchBinaryGet().format(),  
        zwave.switchMultilevelV1.switchMultilevelGet().format(),  
        zwave.meterV2.meterGet(scale: 0).format(), // get kWh  
        zwave.meterV2.meterGet(scale: 2).format(), // get Watts  
        zwave.sensorMultilevelV1.sensorMultilevelGet().format(),  
        zwave.sensorMultilevelV5.sensorMultilevelGet(sensorType:1, scale:1).format(), // get  
        zwave.batteryV1.batteryGet().format(),  
        zwave.basicV1.basicGet().format(),  
    ], 1200)  
}  
  
// If you add the Polling capability to your device type, this command  
// will be called approximately every 5 minutes to check the device's state  
def poll() {  
    zwave.basicV1.basicGet().format()  
}  
  
// If you add the Configuration capability to your device type, this  
// command will be called right after the device joins to set  
// device-specific configuration commands.  
def configure() {  
    delayBetween([  
        // Note that configurationSet.size is 1, 2, or 4 and generally  
        // must match the size the device uses in its configurationReport  
        zwave.configurationV1.configurationSet(parameterNumber:1, size:2, scale:Configuration)  
  
        // Can use the zwaveHubNodeId variable to add the hub to the  
        // device's associations:  
        zwave.associationV1.associationSet(groupingIdentifier:2, nodeId:zwaveHubNodeId).forma  
  
        // Make sure sleepy battery-powered sensors send their  
        // WakeUpNotifications to the hub every 4 hours:  
        zwave.wakeUpV1.wakeUpIntervalSet(seconds:4 * 3600, nodeId:zwaveHubNodeId).format(),  
    ])  
}
```

ZigBee Primer

Before we start, let's take a look at a full ZigBee message as it would look in a SmartThings Device Handler. Then we'll break up the message into its parts and dive into what each part means. Make sure you download the ZigBee Cluster Library as a reference for ZigBee message formatting and what is possible for each device. You can also see our *ZigBee Reference* (page 1046) for more detailed descriptions of our library methods.

Here are some full commands:

Set the level of a device `zigbee.command(0x0008, 0x04, "FE0500")`

Read the current level (e.g. of a light) `zigbee.readAttribute(0x0008, 0x0000)`

Write the value 0xBEEF to cluster 0x0008 attribute 0x0010 `zigbee.writeAttribute(0x0008, 0x0010, DataType.UINT16, 0xBEEF)`

Report battery level every 10 minutes to 6 hours if it changes value by 1 `zigbee.configureReporting(0x0001, 0x0021, DataType.UINT8, 600, 21600, 0x01)`

The 4 Main types of ZigBee Messages

- `zigbee.command` - A ZigBee command for a given cluster.
- `zigbee.readAttribute` - A ZigBee Read Attribute requesting the value of an attribute from a cluster.
- `zigbee.writeAttribute` - A ZigBee Write Attribute writing a value to the attribute of a cluster.
- `zigbee.configureReporting` - A ZigBee Configure Report that configures a cluster attribute to report changes of a given amount within a certain time period.

139.1 Device Network ID

All connected devices have a Device Network ID that is used to route messages correctly to the device. In the loosest terms think of the Network ID as the IP Address. It is a 4 digit hex number that the device gets while pairing. Since the Network ID is unique by device on a network, it can be handled by the ZigBee library provided by SmartThings and needs not be handled directly.

139.2 Endpoints

Endpoints are simple. Think of them basically as ports. Different endpoints can support different clusters and a device can have multiple endpoints to do different things. Endpoints can be used to separate functionality when needed. For example a temperature sensor can have the Temperature Measurement Cluster on endpoint 1 and have Over The Air Boot loader Cluster on endpoint 2.

139.3 Clusters

Clusters are a group of commands and attributes that define what a device can do. Think of clusters as a group of actions by function. A device can support multiple clusters to do a whole variety of tasks. The majority of clusters are defined by the ZigBee Alliance and listed in the ZigBee Cluster Library. There are also profile specific clusters that are defined by their own ZigBee profile like Home Automation or ZigBee Smart Energy, and Manufacture Specific clusters that are defined by the manufacture of the device. These are typically used when no existing cluster can be used for a device.

Most used clusters are

- 0x0006 - On/Off (Switch)
 - 0x0008 - Level Control (Dimmer)
 - 0x0201 - Thermostat
 - 0x0202 - Fan Control
 - 0x0402 - Temperature Measurement
 - 0x0406 - Occupancy Sensing
-

139.4 Commands

Commands are basically actions a device can take. It's how we get things to do stuff. Commands and whats available are defined by the cluster.

Keeping on the On/Off cluster as an example, the available commands are:

- 0x00 - Off
- 0x01 - On
- 0x02 - Toggle

In a SmartThings Device Type the following line would turn a switch off (look at the last number):
`zigbee.command(0x0006, 0x00)`

This would turn it on: `zigbee.command(0x0006, 0x01)`

This would toggle it: `zigbee.command(0x0006, 0x02)`

139.5 Read and Write Attributes

Attributes are used to get information from a device and to set preferences on a device. The two main types are Read and Write. The data type and values are specified by cluster.

An example of a Read Attribute that would read the current level of a dimmer and return the value:

```
zigbee.readAttribute(0x0008, 0x0000)
```

Write Attributes are used to set specific preferences. Write attributes can need specific data type that the payload is in.

An example of a Write Attribute that would set the transition time from on to off of a dimmer look like this:

```
zigbee.writeAttribute(0x0008, 0x0010, DataType.UINT16, 0x0014)
```

In this case the value (0x0014) translates to 2 seconds. Breaking the payload down we see that the hex value of 0x0014 equals the decimal value of 20. $20 * 1/10$ of a second equals 2 seconds.

139.6 Configure reporting

Many times you will have an attribute for a given device that you are interested in receiving notifications about. For example you may want to be notified any time the battery level changes. The way to do this in ZigBee is by configuring a report for that cluster.

An example of configuring a report for the battery level: `zigbee.configureReporting(0x0001, 0x0021, DataType.UINT8, 600, 21600, 0x01)`

This is for cluster 0x0001 (power cluster), attribute 0x0021 (battery level), whose type is UINT8, the minimum time between reports is 10 minutes (600 seconds) and the maximum time between reports is 6 hours (21600 seconds), and the amount of change needed to trigger a report is 1 unit (0x01).

139.7 Device discovery

After a ZigBee device joins the network it must be queried in order to select the correct Device Handler. After a device joins (or rejoins) the network the Hub will collect the simple descriptor, manufacturer, model and application version for each endpoint without any interaction with the cloud. The Hub will automatically resend any messages that the device does not respond to in a timely manner. Once all the information has been obtained it is sent to the cloud in the `zbjoin` message. This message is visible in Hub Events.

Here is an example of the message when a SmartSense Multi Sensor was joined:

```
zbjoin: {"dni": "5CF4",
  "d": "000D6F0005767F37",
  "capabilities": "80",
  "endpoints": [{"simple": "01 0104 0402 00 07 0000 0001 0003 0020 0402 0500 0B05 01 0019",
    "application": "",
    "manufacturer":
      "Centralite",
    "model": "3325-S"},
    {"simple": "02 C2DF 0107 00 05 0000 0001 0003 0B05 FC46 01 0003",
    "application": "",
    "manufacturer": null,
    "model": null}
  ]
}
```

The value is a dictionary that contains all the information gathered from the device. Here is what each part means:

- dni: *Device Network ID* (page 523)
- d: the ZigBee EUID aka long address
- capabilities: the MAC capability field from the Device Announce message (not currently used by SmartThings)
- endpoints: a list of information for each available endpoint
- simple: a space separated string of hex values that contains the following pieces of information:
 - Endpoint
 - Profile ID
 - Device ID
 - Device version
 - Number of in/server clusters
 - List of In/server clusters
 - Number of out/client clusters
 - List of out/client clusters
- application: the Application Version read from attribute 0x0001 of the Basic Cluster
- manufacturer: The Manufacturer value read from attribute 0x0004 of the Basic Cluster
- model: The Model value read from attribute 0x0005 of the Basic Cluster

See *ZigBee fingerprinting* (page 469) for more information on how the platform uses this information to find the correct Device Handler for the device.

139.8 Useful ZigBee references

ZigBee Cluster Library (ZCL)

ZigBee Home Automation (HA)

ZigBee Specification

Building ZigBee Device Handlers

Note: If you are integrating a new ZigBee switch or bulb with SmartThings, see the *Using the ZigBee Device Form* (page 529) section below to learn how you can integrate these devices without the need to write code.

140.1 Commands

SmartThings provides a library to make working with ZigBee easier. Every Device Handler has a reference to this library injected into it, with the name `zigbee`.

This library will be used in the examples below. You can see the *ZigBee Reference* (page 1046) for more detailed documentation.

There are four common ZigBee commands that you will use to integrate SmartThings with your ZigBee Devices.

140.1.1 Read

Read gets the devices current state and is formatted like this:

```
def refresh() {
    zigbee.readAttribute(0x0B04, 0x050B)
}
```

In this example, the device type (from the “CentraLite Switch” device type) is calling the “refresh” function. It is sending a ZigBee Read Attribute request via the `readAttribute()` method to read the current state (the active power draw). The cluster we are reading here is Electrical Measurement (0xB04) and specifically the Active Power Attribute (0x50B).

Component	Description
0x0B04	Cluster
0x050B	Attribute

140.1.2 Write

Write sets an attribute of a ZigBee device and is formatted like this:

```
def configure() {
    zigbee.writeAttribute(8, 0x10, 0x21, 0x0014)
}
```

In this example (from the “ZigBee Dimmer” Device Handler) we are writing to an attribute to set the amount of time it takes for a light to fully dim on and off. Here we are using the Level Control Cluster (8) to write to the attribute that defines on and off transition time (0x10). The value we are using is formatted in an Unsigned 16-bit integer (0x21) with the payload being in 1/10th of a second. In this case the payload ({0014}) translates to 2 seconds. Breaking the payload down we see that the hex value of 0x0014 equals the decimal value of 20. $20 * 1/10$ of a second equals 2 seconds.

Each attribute possesses a specific data type. The corresponding value for this data type can be found in table 2.16 of the [ZigBee Cluster Library](#).

Note: The payload in the example above, {0014}, is a hex string. The length of the payload must be two times the length of the data type. For example, if the datatype is 16-bit, then the payload should be 4 hex digits.

Component	Description
8	Cluster
0x10	Attribute Set
0x21	Data Type
0x0014	Payload

140.1.3 Command

Command invokes a command on a ZigBee device and is formatted like this:

```
def on() {
    zigbee.command(0x0006, 0x01)
}
```

In this example (from the “ZigBee Dimmer” device type) we are sending a ZigBee Command to turn the device on. We use the On/Off Cluster (6) and send the command to turn on (1). This commands has no payload, so we exclude it from the passed in parameters.

Component	Description
0x0006	Cluster
0x01	Command

140.1.4 Configure

Configure reporting instructs a device to notify us when an attribute changes and is formatted like this:

```
def configure() {
    configureReporting(0x0006, 0x0000, 0x10, 0, 600, null)
}
```

In this example (using the “Centralite Switch” Device Handler), the bind command is sent to the device using its Network ID which can be determined using `0x${device.deviceNetworkId}`. Then using source and destination endpoints for the device and Hub (1 1), we bind to the On/Off Clusters (6) to get Events from the device. The last part of the message contains the Hub’s ZigBee id which is set as the Location for the device to send callback messages to. Note that not all devices support binding for Events.

Component	Description
0x0006	Cluster
0x0000	Attribute ID
0x10	Boolean data type
0	Minimum report time
600	Maximum report time
null	Reportable change (discrete)

140.2 ZigBee utilities

In order to work with ZigBee you will need to use the ZigBee Cluster Library extensively to look up the proper values to send back and forth to your device. You can download this document [here](#).

There is also a ZigBee utility class covered in the *ZigBee Reference* (page 1046).

140.3 Best practices

- The use of ‘raw ...’ commands is deprecated. Instead use the documented methods on the ZigBee library. If you need to do something that requires the use of a ‘raw’ command let us know and we will look at adding it to the ZigBee library.
 - Do not use `sendEvent()` in command methods. Sending Events should be handled in the `parse` method.
-

140.4 Using the ZigBee Device Form

To integrate a new ZigBee switch or bulb with SmartThings, you can use the *From ZigBee Device Form*.

From Form From Code From Template From ZigBee Device Form

If you are interested in integrating a new ZigBee bulb or switch with SmartThings you might not need to do any coding at all. This form allows you to create fully functional ZigBee devices just by entering basic information about the device. Filling out and submitting the form adds a fingerprint for your device to the appropriate existing Device Handler. You can then test the pairing and function of your device and submit the update as a Publication Request.

ZigBee Application Profile ZigBee Home Automation (HA) Home Automation and Light Link supported

ZigBee Device: ZigBee Switch What option best describes the device that you are integrating with SmartThings

ZigBee Manufacturer Name The name specified in Basic Cluster (0x0000), Attribute (0x0004)

ZigBee Model Name The name specified in Basic Cluster (0x0000), Attribute (0x0005)

Device Join Name The marketing name of the device that you want to show when it pairs in the SmartThings App

ZigBee Server Clusters Supported Comma separated clusters values (inClusters)

ZigBee Client Clusters Supported (optional) Comma separated cluster values (outClusters)

Device ID (optional) Device ID

Create Cancel

140.4.1 What it does

By entering the ZigBee information for the device in the form, the appropriate existing Device Handler will be updated with the device’s fingerprint.

140.4.2 Use it if

- You are the device manufacturer, or otherwise have access to the required ZigBee device information requested on the form.
- The device is best described as one of the following:
 - ZigBee Switch
 - ZigBee Switch with Power
 - ZigBee Dimmer/Bulb
 - ZigBee Dimmer/Bulb with Power
 - ZigBee Color Temperature Bulb

140.4.3 How to use

Simply fill out the required fields in the form with the information for the device, and click *Create*.

You will then see the updated Device Handler code in the IDE editor. You can then test that your device pairs with SmartThings and functions as expected, and then make an update as a Publication Request.

ZigBee Example

An example of a ZigBee device-type is a ZigBee dimmer.

Here is the code.

```
/**
 * Copyright 2015 SmartThings
 *
 * Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except
 * in compliance with the License. You may obtain a copy of the License at:
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software distributed under the License
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See
 * for the specific language governing permissions and limitations under the License.
 */

metadata {
    definition (name: "ZigBee Dimmer", namespace: "smarththings", author: "SmartThings") {
        capability "Actuator"
        capability "Configuration"
        capability "Refresh"
        capability "Switch"
        capability "Switch Level"

        fingerprint profileId: "0104", inClusters: "0000, 0003, 0004, 0005, 0006, 0008"
        fingerprint profileId: "0104", inClusters: "0000, 0003, 0004, 0005, 0006, 0008, 0B04, FCOF",
        fingerprint profileId: "0104", inClusters: "0000, 0003, 0004, 0005, 0006, 0008, FF00", outCl
        fingerprint profileId: "0104", inClusters: "0000, 0003, 0004, 0005, 0006, 0008, 0B05", outCl
    }

    tiles(scale: 2) {
        multiAttributeTile(name:"switch", type: "lighting", width: 6, height: 4, canChangeIcon: true) {
            tileAttribute ("device.switch", key: "PRIMARY_CONTROL") {
                attributeState "on", label:'${name}', action:"switch.off", icon:"st.switches.light.on"
                attributeState "off", label:'${name}', action:"switch.on", icon:"st.switches.light.off"
                attributeState "turningOn", label:'${name}', action:"switch.off", icon:"st.switches.light.turningOn"
                attributeState "turningOff", label:'${name}', action:"switch.on", icon:"st.switches.light.turningOff"
            }
            tileAttribute ("device.level", key: "SLIDER_CONTROL") {
                attributeState "level", action:"switch_level.setLevel"
            }
        }
    }
}
```

```
    }
    }
    standardTile("refresh", "device.switch", inactiveLabel: false, decoration: "flat", width: 2,
        state "default", label:"", action:"refresh.refresh", icon:"st.secondary.refresh"
    )
    main "switch"
    details(["switch", "refresh"])
}
}

// Parse incoming device messages to generate events
def parse(String description) {
    log.debug "description is $description"

    def event = zigbee.getEvent(description)
    if (event) {
        sendEvent(event)
    }
    else {
        log.warn "DID NOT PARSE MESSAGE for description : $description"
        log.debug zigbee.parseDescriptionAsMap(description)
    }
}

def off() {
    zigbee.off()
}

def on() {
    zigbee.on()
}

def setLevel(value) {
    zigbee.setLevel(value)
}

def refresh() {
    return zigbee.readAttribute(0x0006, 0x0000) +
        zigbee.readAttribute(0x0008, 0x0000) +
        zigbee.configureReporting(0x0006, 0x0000, 0x10, 0, 600, null) +
        zigbee.configureReporting(0x0008, 0x0000, 0x20, 1, 3600, 0x01)
}

def configure() {
    log.debug "Configuring Reporting and Bindings."

    return zigbee.configureReporting(0x0006, 0x0000, 0x10, 0, 600, null) +
        zigbee.configureReporting(0x0008, 0x0000, 0x20, 1, 3600, 0x01) +
        zigbee.readAttribute(0x0006, 0x0000) +
        zigbee.readAttribute(0x0008, 0x0000)
}
}
```

Other Useful Methods

Device Handlers have available to them other APIs for features like scheduling, storing data, and making HTTP requests. While not often necessary for Device Handlers, these features can be useful for certain use cases.

142.1 Scheduling

Device Handlers can schedule future executions, just like SmartApps.

You can learn about scheduling in the *Scheduling* (page 345) guide.

142.2 Storing data

Device Handlers can persist small amounts of data across executions using `state`, just as SmartApps. Note that Atomic State is **not available** to Device Handlers.

You can learn about storing data in the *Storing Data With State* (page 315) guide.

142.3 Making external HTTP requests

Device Handlers can make HTTP requests to third party services, just like SmartApps.

- *Making Synchronous External HTTP Requests* (page 365)
- *Making Asynchronous External HTTP Requests (Beta)* (page 371)

Device Certification Overview

Because we offer an open platform, a wide range of devices can be certified to work with SmartThings. Currently, anybody can submit a device for certification at no cost. Certifying your device will provide a great experience for users, meaning that your device works seamlessly with the rest of the SmartThings platform.

Examples of devices already certified to work with SmartThings can be viewed [here](#).



The device certification process consists of the following steps:

1. **Create** a virtual representation of your device using a *Device Handlers* (page 449)
2. **Test** the Device Handler by publishing it to your account and pairing your device with your Hub
3. Once you've successfully tested your Device Handler, **submit** it for publication
4. The SmartThings certification team will contact you about how to **ship** your device to us and complete the certification process

We're always looking for ways to improve and shorten the time it takes to certify devices. Stay tuned for future improvements!

Part XIII

Cloud- and LAN-connected Devices

Cloud- and LAN-connected devices are devices that use either a third-party service, like the Ecobee thermostat, or communicate over the LAN (local area network) like the Sonos system. These devices require a unique implementation of their Device Handlers. Cloud- and LAN-connected devices use a Service Manager SmartApp along with a Device Handler for authentication, maintaining connections, and device communications. This guide will walk you through Service Manager and Device Handler creation for both of these scenarios.

Table of Contents:

Service Manager Design Pattern

144.1 Basic overview

Devices that connect through the internet as a whole (cloud) or LAN devices (on your local network) require a defined Service-Manager SmartApp, in addition to the usually expected Device Handler. The Service Manager makes the connection with the device, handling the input and output interactions, and the Device Handler parses messages.

144.2 Cloud-connected devices

When using a Cloud-connected device, the service manager is used to discover and initiate a connection between the device and your Hub, using OAuth connections to external third parties. Then the Device Handler uses this connection to communicate between the Hub and device.

144.3 LAN-connected devices

When using a LAN-connected device, the service manager is used to discover and initiate a connection between the device and your Hub, using the protocols SSDP or mDNS/DNS-SD. Then the device-handler uses UPnP/SOAP Calls or REST Calls to communicate outgoing messages between the Hub and device.

Building Cloud-connected Device Types

Cloud-connected devices use a third-party service to accomplish device communication. An example of such a device is the Ecobee thermostat.

When developing a Device Handler for a Cloud-connected device, you must create a Service Manager SmartApp that will handle authenticating with the third-party service, communicating with the device, and reacting to any device changes that occur.

This guide overviews the concept of the Service Manager/Device Handler architecture and also gives an example of both the Service Manager and Device Handler creation.

Table of Contents:

145.1 Division of Labor

The Cloud-connected device paradigm consists of a Service Manager and Device Handlers. The purpose of this guide is to introduce you to the core concepts of Cloud-connected device development, and provide some examples to help you get started.

145.1.1 Service Manager responsibilities

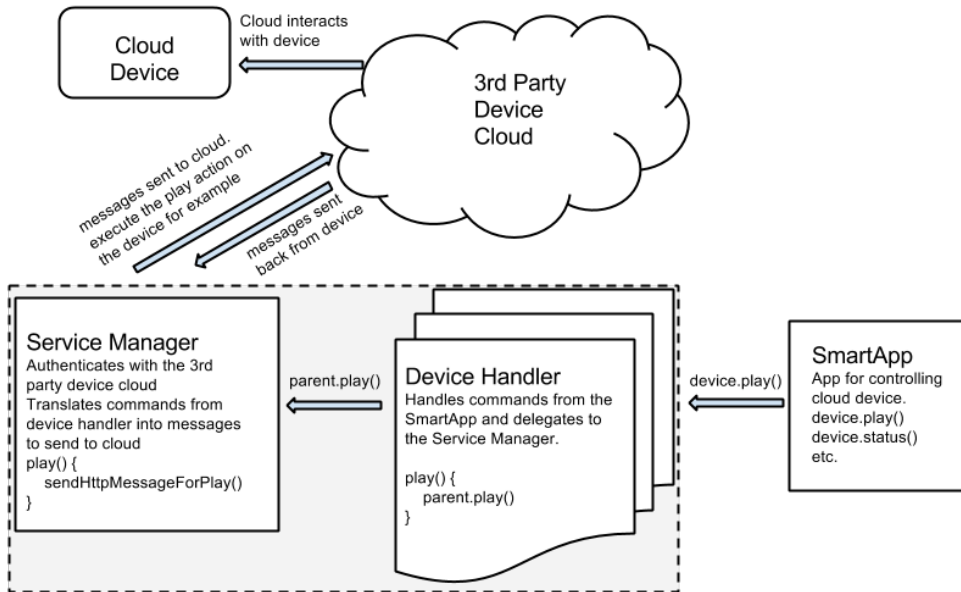
The service manager is responsible for the discovery of the devices. It sends out a request to a third party cloud and parses through the response, finding just the devices you are looking for. Upon discovery, it allows you to add device(s) that it has found. From there, it saves your connection to be able to make future interactions with the device.

145.1.2 Device Handler responsibilities

The Device Handler is responsible for creating and receiving device specific messages, and allowing them to work within the SmartThings infrastructure. It takes in a SmartApp specific command and outputs device specific commands to be passed to the cloud. It also allows you to subscribe to responses from the device and trigger other commands as needed.

145.1.3 How it all works

The following depiction gives a general overview of how a Cloud-connected device works. Take note of the Service Manager and Device Handler. We will dive into how to build these next.



145.2 Building the Service Manager

The Service Manager’s responsibilities are:

- To authenticate with the third-party cloud service.
- Device discovery.
- Add/Change/Delete device actions.
- Handle sending any messages that require the authentication obtained.

Below we will get into the details of what is outlined above. First, let’s see an illustration of it in a SmartThings application using the Ecobee Thermostat.

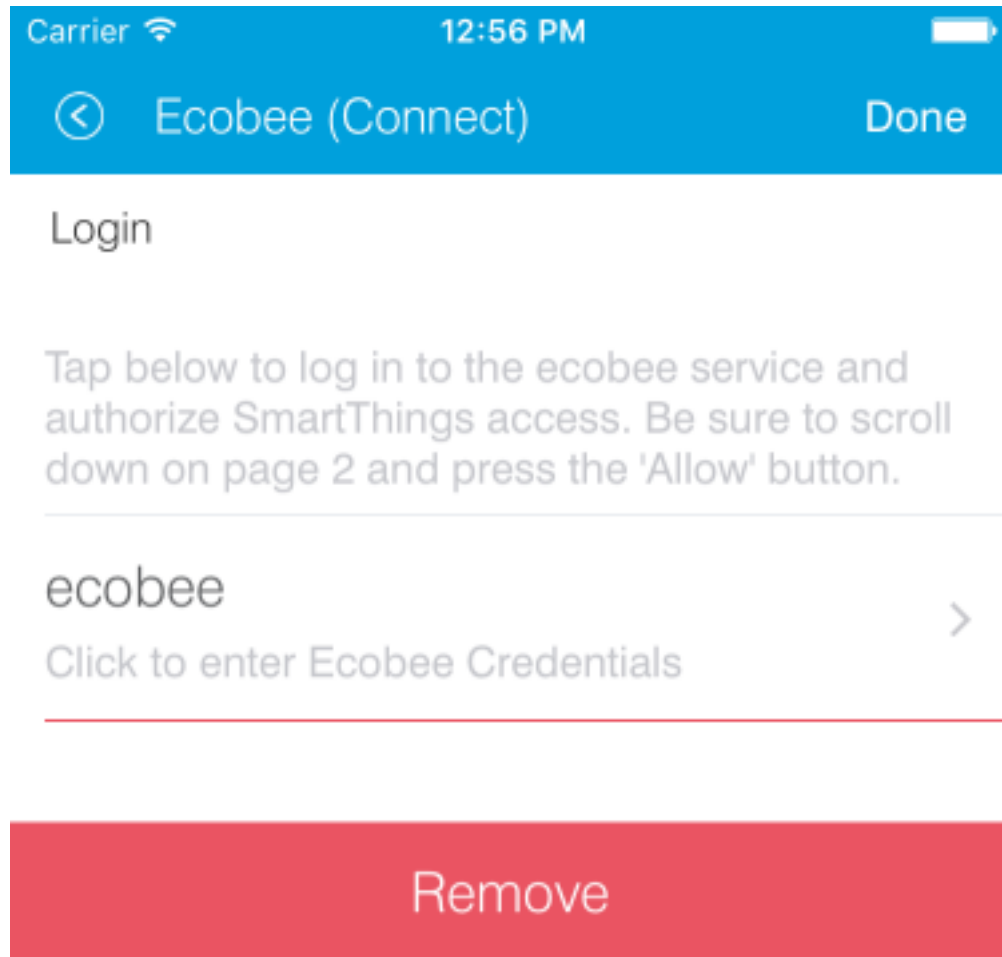
145.2.1 Authentication using OAuth

Any Service Manager authenticated with a third party via OAuth must itself have OAuth enabled. This is because eventually the third-party service will call back into the SmartApp and hit the `/oauth/initialize` and `/oauth/callback` endpoints.

End user experience

As an end user you start by selecting the Service Manager SmartApp for Ecobee Thermostat from the SmartApps screen of the SmartThings mobile app.

Authorization with the third party is the first part of the configuration process. You will be taken to a page that describes how the authorization process works.



From this screen you will then be directed to the third-party site, i.e., the Ecobee Thermostat site in this case, embedded within the SmartThings mobile application. Here you will enter your Ecobee Thermostat service username and password.

Next this third-party Ecobee server will show you what access permissions SmartThings will have to your Ecobee account. It also gives you an opportunity to accept or decline.

After you accept, on the following screen you will finish the configuration by tapping on the *Done* icon on the top right.

Next, you will be taken back to the initial configuration screen where you select your device to complete the installation.

Carrier 12:57 PM

ecobee Done

ecobee

Log in to Authorize App

Email Address

Password

ecobee

Authorize App



You are authorizing "*SmartApp2*", provided by *SmartThings 2*.

This App will have permissions to view the thermostat settings and data in your account.

This App is not provided, tested or warranted by ecobee. You agree to use this App at your own risk.

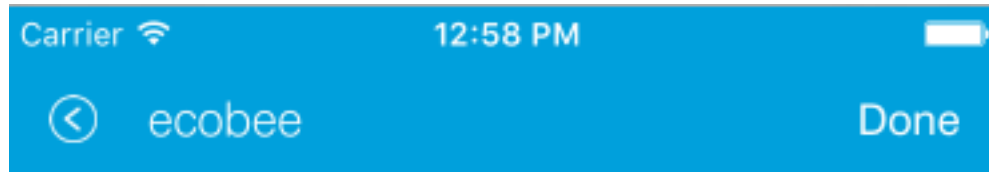
Application Summary:

Connect to SmartThings

Application Description:

Decline

Accept



Your ecobee Account is now
connected to SmartThings!

Click 'Done' to finish setup.

Implementation

OAuth is an industry standard for authentication. However, the third-party service may use a different standard. In that case, consult their documentation and implement it. The basic concepts will be similar to that of OAuth. The below example will walk through what is necessary for OAuth authentication.

There are two endpoints that all Service Manager SmartApps must define.

```
mappings {
  path("/oauth/initialize") {action: [GET: "oauthInitUrl"]}
  path("/oauth/callback") {action: [GET: "callback"]}
}
```

The `/oauth/initialize` endpoint will be called during initialization. This endpoint will then forward the user to the third-party service so they can log in.

The third-party service will be redirected to the `/oauth/callback` endpoint after the authentication has been successful. Usually this is where the call is made to the third-party service to exchange an authorization code for an access token.

The overall idea is this:

- You will create a page on Service Manager SmartApp that will call out to the third-party API to initiate the authentication.
- The end result is an access token that SmartThings platform will then use to communicate with the third-party API.

In your Service Manager SmartApp preferences you create a page for authorization.

```
preferences {
  page(name: "Credentials", title: "Sample Authentication", content: "authPage", nextPage: "sample")
  ...
}
```

The `authPage` method will perform the following tasks:

- Create a SmartApp access token that will be sent to the third party so that the third party can call back into SmartThings SmartApp.
- Check to make sure an access token doesn't already exist for this particular third-party service.
- Initialize the OAuth flow with the third party service if there is no access token.

Let's take a look at how we would accomplish this with `authPage()`.

```
def authPage() {
  // Check to see if SmartApp has its own access token and create one if not.
  if(!state.accessToken) {
    // the createAccessToken() method will store the access token in state.accessToken
    createAccessToken()
  }

  def redirectUrl = "https://graph.api.smartthings.com/oauth/initialize?appId=${app.id}&access_token=${state.accessToken}"
  // Check to see if SmartThings already has an access token from the third-party service.
  if(!state.authToken) {
    return dynamicPage(name: "auth", title: "Login", nextPage: "", uninstall: false) {
      section() {
        paragraph "tap below to log in to the third-party service and authorize SmartThings a"
        href url: redirectUrl, style: "embedded", required: true, title: "3rd Party product",
      }
    }
  }
}
```

```

    } else {
        // SmartThings has the token, so we can just call the third-party service to list our devices
    }
}

```

There are a few things worth noting here:

- First, we are using `state` to store our tokens. Your specific needs may be different depending on your implementation. To learn more about how `state` works and what your options are, visit the [Storing Data With State](#) (page 315) guide.
- If we do not have a token from the third-party service, we start the OAuth flow by calling the SmartThings `initialize` endpoint. This is a static endpoint that will store a few bits of information about your SmartApp, such as the `id`, and forwards the request to the `/oauth/initialize` endpoint defined in the SmartApp.

Initialize endpoint

This endpoint is used to initialize the OAuth flow to a third-party service. The `/oauth/initialize` endpoint will save all the query parameters passed to it, but requires the following three parameters:

- The SmartApp ID,
- The SmartApp's access token, and
- The installed URL of the SmartApp. The endpoint will then call the mapped `/oauth/initialize` endpoint defined in the SmartApp with all the query parameters passed to it.

```
https://graph.api.smarthings.com/oauth/initialize
```

Required parameters	Value
<code>appId</code>	The SmartApp ID
<code>access_token</code>	The SmartApp's access token
<code>apiServerUrl</code>	The URL of the server that the SmartApp is installed on. This information can be retrieved with the <code>getApiServerUrl()</code> method call.

Example:

```
def redirectUrl = "https://graph.api.smarthings.com/oauth/initialize?appId=${app.id}&access_token=${
```

The `initialize` endpoint will forward the mapping defined in SmartApp to the `/oauth/initialize`. This method will be responsible for redirecting the user to the third-party login page. Below is an example of how it works:

```

def oauthInitUrl() {

    // Generate a random ID to use as a our state value. This value will be used to verify the response
    state.oauthInitState = UUID.randomUUID().toString()

    def oauthParams = [
        response_type: "code",
        scope: "smartRead,smartWrite",
        client_id: appSettings.clientId,
        client_secret: appSettings.clientSecret,
        state: state.oauthInitState,
        redirect_uri: "https://graph.api.smarthings.com/oauth/callback"
    ]

    redirect(location: "${apiEndpoint}/authorize?${toQueryString(oauthParams)}")
}

```

```
// The toQueryString implementation simply gathers everything in the passed in map and converts them
String toQueryString(Map m) {
    return m.collect { k, v -> "${k}=${URLEncoder.encode(v.toString())}" }.sort().join("&")
}
```

The `oauthInitUrl()` method sets up a request used to present the user with the third-party login page. Often the third-party service will require information passed along with this request as query parameters. The actual parameters sent with the request will vary depending on what the third-party service expects, so consult their API documentation to find specifics.

We are expecting to get an authorization code as a result of this request. We will later exchange this authorization code for an access token. We will create the access token request in our callback handler as seen below. But for now, let's look at some basic parameters usually associated with authorization code requests.

Common parameters	Value
response_type	The type of authorization defined by third-party service. Usually code or token.
scope	Defines the scope of the request, i.e., what actions will be performed.
client_id	The client ID issued by the third-party service when signing up for access to their API. A best practice is to configure this parameter as an app setting in your SmartApp.
client_secret	The client secret issued by the third-party service when signing up for access to their API. A best practice is to configure this parameter as an app setting in your SmartApp.
state	Usually the <code>state</code> is not required, but is used to track state across requests. We will use this to validate the response we get back from the third party.
redirect_uri	The URI to be redirected to after the user has successfully authenticated with the third-party service. Usually this information is requested when signing up with the third-party service. This parameter must match what was entered at that time. For SmartApp development, this should always be the static value: <code>https://graph.api.smartthings.com/oauth/callback.</code>

Callback endpoint

The third-party service will redirect the user to the callback endpoint after the user has been successfully authenticated. For SmartApp development, this should always be the static value: `https://graph.api.smartthings.com/oauth/callback`. The callback endpoint is typically where the authorization code—that was acquired from the initialization—will be used to request the access token. Let's look at an example.

```
def callback() {
    log.debug "callback() >> params: $params, params.code ${params.code}"

    def code = params.code
    def oauthState = params.state

    // Validate the response from the third party by making sure oauthState == state.oauthInitState
    if (oauthState == state.oauthInitState) {
        def tokenParams = [
            grant_type: "authorization_code",
            code       : code,
            client_id  : appSettings.clientId,
            client_secret: appSettings.clientSecret,
            redirect_uri: "https://graph.api.smartthings.com/oauth/callback"
        ]
    }
}
```

```

// This URL will be defined by the third party in their API documentation
def tokenUrl = "https://www.someservice.com/home/token?${toQueryString(tokenParams)}"

httpPost(uri: tokenUrl) { resp ->
    state.refreshToken = resp.data.refresh_token
    state.authToken = resp.data.access_token
}

if (state.authToken) {
    // call some method that will render the successfully connected message
    success()
} else {
    // gracefully handle failures
    fail()
}

} else {
    log.error "callback() failed. Validation of state did not match. oauthState != state.oauthIn
}
}

// Example success method
def success() {
    def message = """
        <p>Your account is now connected to SmartThings!</p>
        <p>Click 'Done' to finish setup.</p>
    """
    displayMessageAsHtml(message)
}

// Example fail method
def fail() {
    def message = """
        <p>There was an error connecting your account with SmartThings</p>
        <p>Please try again.</p>
    """
    displayMessageAsHtml(message)
}

def displayMessageAsHtml(message) {
    def html = """
        <!DOCTYPE html>
        <html>
            <head>
            </head>
            <body>
                <div>
                    ${message}
                </div>
            </body>
        </html>
    """
    render contentType: 'text/html', data: html
}

```

In this callback we first check to make sure that the state returned from the authorization code request matches what we sent as the state. This is how we know that the response is intended for us. If it matches, we then set up the parameters for the access token request. Common parameters are as follows:

Common parameters	value
grant_type	This is the type of grant we are requesting. The third-party service will define the expected value.
code	The authorization code we obtained in the previous request.
client_id	The same client_id that we used in the previous request, which was issued by the third-party service.
client_secret	The same client_secret that we used in the previous request, which was issued by the third-party service.
redirect_uri	The same redirect_uri that we used in the previous request. This will usually be verified by the third-party service.

We issue an HTTP POST request to get the token. If we receive a success response, we will save the access token that was issued by the third-party service, along with the refresh token, in `state`.

Once we have acquired the access token, our authentication process is complete. Usually the next step is to display some message to the end user about the success of the operation.

Important: `revokeAccessToken()` should be called when the SmartApp's access token is no longer required. This is true when a user uninstalls the SmartApp. It is also a good practice to revoke the access token after successful authentication with the 3rd party, unless the token will be used to access other endpoints in your SmartApp.

Refreshing the OAuth token

OAuth tokens are available for a finite amount of time, so you will often need to account for this, and if needed, refresh your `access_token`. Above we illustrated how we initiate the request for the access and refresh tokens, and how we saved them in our SmartApp. If we make a request to the third-party service API and get an “expired token” response, it is up to us to issue a new request to refresh the access token. This is where the refresh token comes into play.

If you run an API request and your `access_token` is determined invalid, for example:

```
if (resp.status == 401 && resp.data.status.code == 14) {
  log.debug "Storing the failed action to try later"
  def action = "actionCurrentlyExecuting"
  log.debug "Refreshing your auth_token!"
  refreshAuthToken()
  // replay initial request from the action variable
  retryInitialRequest(action)
}
```

you can use your `refresh_token` to get a new `access_token`. To do this, you just need to post to a specified endpoint and handle the response properly.

```
private refreshAuthToken() {
  def refreshParams = [
    method: 'POST',
    uri: "https://api.thirdpartysite.com",
    path: "/token",
    query: [grant_type:'refresh_token', code:"${state.sampleRefreshToken}", client_id:XXXXXXX],
  ]
  try{
    def jsonMap
    httpPost(refreshParams) { resp ->
      if(resp.status == 200)
      {
```

```
        jsonMap = resp.data
        if (resp.data) {
            state.sampleRefreshToken = resp?.data?.refresh_token
            state.sampleAccessToken = resp?.data?.access_token
        }
    }
}
```

There are some outbound connections in which we are using OAuth to connect to a third party device cloud (Ecobee, Quirky, Jawbone, etc). In these cases it is the third-party device cloud that issues an OAuth token to SmartThings so that SmartThings can call their APIs.

However, these same third-party device clouds also support webhooks and subscriptions that allow SmartThings to receive notifications when something changes in their cloud.

In this case, *and ONLY in this case*, the Service Manager SmartApp issues its own OAuth token and embeds it in the callback URL, as a way to authenticate the post backs from the external cloud.

145.2.2 Discovery

Identifying devices in the third-party device cloud

The techniques you will use to identify devices in the third-party cloud will vary, because you are interacting with unique third-party APIs which all have unique parameters. Typically you will authenticate with the third-party API using OAuth; then call an API-specific method. For example, it could be as simple as this:

```
def deviceListParams = [
    uri: "https://api.thirdpartysite.com",
    path: "/get-devices",
    requestContentType: "application/json",
    query: [token:"XXXX",type:"json" ]
]

httpGet(deviceListParams) { resp ->
    //Handle the response here
}
```

Creating child devices

Within a Service Manager SmartApp, you create child devices for all your respective cloud devices.

```
settings.devices.each { deviceId->
    def device = state.devices.find{it.id==deviceId}
    if (device) {
        def childDevice = addChildDevice("smarththings", "Device Name", deviceId, null, name: "Device")
    }
}
```

Getting initial device state

Upon initial discovery of a device, you need to get the state of your device from the third-party API. This would be the current status of various attributes of your device. You need to have a method defined in your Service Manager that is responsible for connecting to the API and to check for the updates. You set this method to be called from a

poll method in your Device Handler, and in this case, it is called immediately on initialization. Here is a very simple example which doesn't take into account error checking for the http request.

```
def pollParams = [
  uri: "https://api.thirdpartysite.com",
  path: "/device",
  requestContentType: "application/json",
  query: [format:"json",body: jsonRequestBody]

httpGet(pollParams) { resp ->
  state.devices = resp.data.devices { collector, stat ->
  def dni = [ app.id, stat.identifier ].join('.')
  def data = [
    attribute1: stat.attributeValue,
    attribute2: stat.attribute2Value
  ]
  collector[dni] = [data:data]
  return collector
  }
}
```

145.2.3 Handling adds, changes, deletes

singleInstance Service Manager

Adding the tag `singleInstance: true` to your Service Manager will ensure only one instance of the Service Manager will be installed. All child devices will be installed under the single parent Service Manager. This enforces a one-to-many relationship between the parent Service Manager SmartApp and any child devices.

```
definition(
  name: "Ecobee (Connect)",
  namespace: "smarththings",
  author: "SmartThings",
  description: "Connect your Ecobee thermostat to SmartThings.",
  category: "SmartThings Labs",
  iconUrl: "https://s3.amazonaws.com/smartapp-icons/Partner/ecobee.png",
  iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Partner/ecobee@2x.png",
  singleInstance: true)
```

Implicit creation of new child Devices

When you update your settings in a Service Manager to add additional devices, the Service Manager needs to respond by adding a new device in SmartThings.

```
updated() {
  initialize()
}

initialize() {
  settings.devices.each {deviceId ->
    try {
      def existingDevice = getChildDevice(deviceId)
      if(!existingDevice) {
        def childDevice = addChildDevice("smarththings", "Device Name", deviceId, null, [name
      ]
    } catch (-) {
```

```
        log.error "Error creating device: ${e}"
    }
}
}
```

Implicit removal of child Devices

Similarly when you remove devices in your Service Manager, they need to be removed from SmartThings platform.

```
def delete = getChildDevices().findAll { !settings.devices.contains(it.deviceNetworkId) }
delete.each {
    deleteChildDevice(it.deviceNetworkId)
}
```

Also, when a Service Manager SmartApp is uninstalled, you need to remove its child devices.

```
def uninstalled() {
    removeChildDevices(getChildDevices())
}

private removeChildDevices(delete) {
    delete.each {
        deleteChildDevice(it.deviceNetworkId)
    }
}
```

Note: The `addChildDevice`, `getChildDevices`, and `deleteChildDevice` methods are a part of the *SmartApp* (page 897) API.

Changes in Device name

The device name is stored within the device and you need to monitor if it changes in the third-party cloud.

Explicit delete actions

When a user manually deletes a device in the Things screen on the client device, you need to delete the child device from within the Service Manager.

145.3 Building the Device Handler

The Device Handler for a Cloud-connected device is generally the same as any other Device Handler. The means in which it handles sending and receiving messages from its device is a little bit different. Let's walk through a Cloud-connected Device Handler example.

145.3.1 The Parse method

The parse method for Cloud-connected devices will always be empty. In a Cloud-connected device, Event data is passed down from the Service Manager, not from the device itself, so the parsing is handled in a separate method. The Device Handler doesn't interface directly with a hardware device, which is what parse is used for.

145.3.2 Sending commands to the third-party cloud

Usually the actual implementation of device methods are delegated to its Service Manager. This is because the Service Manager is the entity that has the authentication information. To invoke a method on the parent Service Manager, you simply need to call it in the following format:

```
parent.methodName()
```

As with any other device-type, you need to define methods for all of the possible commands for the capabilities you'd like to support. Then when a user calls this method, it will pass information up to the parent Service Manager, who will make the direct connection to the third party cloud. You might for example want to turn a switch on, so you would call the following.

```
def on() {
    parent.on(this)
}
```

145.3.3 Receiving Events from the third-party cloud

The Device Handler continuously polls the third-party cloud through the service manager to check on the status of devices. When an Event is fired, they can then be passed to the child Device Handler. Note that poll runs every 10 minutes for Service Manager SmartApps.

In the device-type handler:

```
def poll() {
    results = parent.pollChildren()
    parseEventData(results)
}

def parseEventData(Map results){
    results.each { name, value ->
        //Parse events and optionally create SmartThings events
    }
}
```

In the service manager:

```
def pollChildren(){
    def pollParams = [
        uri: "https://api.thirdpartysite.com",
        path: "/device",
        requestContentType: "application/json",
        query: [format:"json",body: jsonRequestBody]
    ]

    httpGet(pollParams) { resp ->
        state.devices = resp.data.devices { collector, stat ->
            def dni = [ app.id, stat.identifier ].join('.')
            def data = [
                attribute1: stat.attributeValue,
                attribute2: stat.attribute2Value
            ]
            collector[dni] = [data:data]
            return collector
        }
    }
}
```

145.3.4 Generating Events at the request of the Service Manager

You won't generate events directly within the Service Manager, but rather request that they are generated within the Device Handler. For example:

In the service manager:

```
childName.generateEvent(data)
```

In the Device Handler:

```
def generateEvent(map results) {
    results.each { name, value ->
        sendEvent(name: name, value: value)
    }
    return null
}
```

Building LAN-connected Device Types

LAN-connected devices communicate with the SmartThings Hub over the LAN. An example of such a device is the Sonos system.

When developing a Device Handler for a LAN device, you must create a service manager SmartApp that will handle discovery of devices on the LAN, in some cases communicate with the device, and react to any device changes that occur via Events.

This guide overviews the concept of the Service Manager/Device Handler architecture and also gives an example of both the Service Manager and Device Handler creation.

Table of Contents:

146.1 Division of Labor

The LAN-connected device paradigm consists of a Service Manager and Device Handlers. The purpose of this guide is to introduce you to the core concepts of LAN-connected device development, and provide some examples to help you get started.

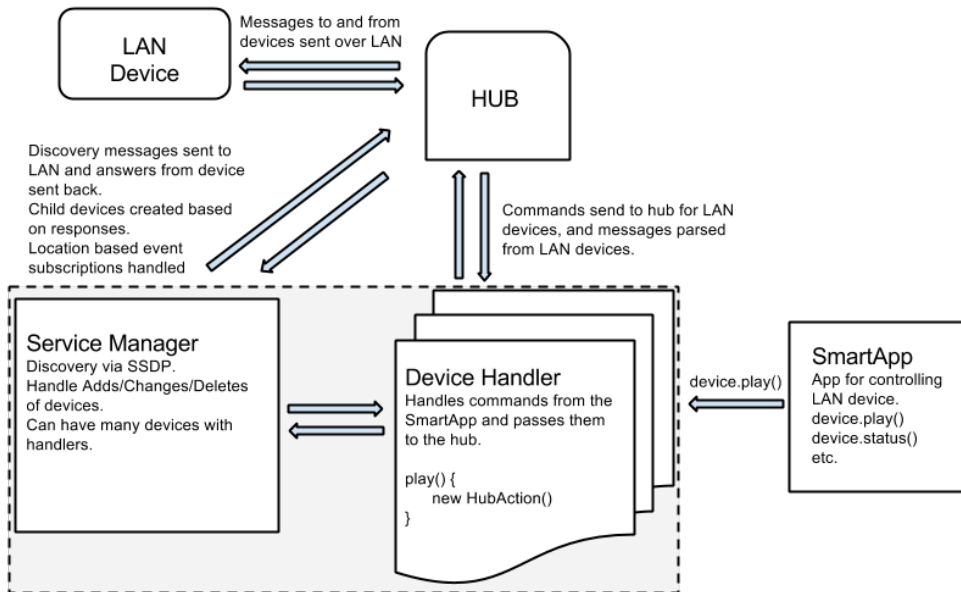
146.1.1 Service Manager responsibilities

The service manager is responsible for the discovery of the devices. It sends out a request and parses through the response, finding just the devices you are looking for. Upon discovery, it allows you to add device(s) that it has found. From there, it saves your connection to be able to make future interactions with the device.

146.1.2 Device Handler responsibilities

The Device Handler is responsible for creating and receiving device specific messages, and allowing them to work within the SmartThings infrastructure. It takes in a SmartApp-specific command and outputs device specific commands. It also allows you to subscribe to responses from the device and trigger other commands as needed.

146.1.3 How it all works



146.2 Building the Service Manager

The Service Manager's responsibilities are:

- **Discovery** - Discover devices on the LAN via SSDP (mDNS/DNS-SD or Bonjour is not currently supported)
- **Verification** - Verify each discovered device through successful fetching of the UPnP device description
- **Inclusion** - Add the device as a child of the service manager
- **Health** - Track IP address and port changes and allow these to make it down to the child device(s) as necessary

Let's take a look at some of the key parts of a Service Manager implementation. The example code referenced throughout this document is derived from the below SmartApp and DeviceType:

```
Generic UPnP Service Manager Generic UPnP Device
```

The above referenced Generic UPnP Device is incomplete. For the complete guide, please see Building the Device Type.

146.2.1 Discovery

Simple Service Discovery Protocol (SSDP) is the main protocol used to find devices on your network. It serves as the backbone of Universal Plug and Play (UPnP), which allows you to easily connect new network devices to a system. See [UPnP Device Architecture 1.1](#) for full specification details.

To discover new devices, you first need to subscribe to Location Events with the correct **search target** for the device. The search target in the below example, **urn:schemas-upnp-org:device:ZonePlayer:1**, is for discovery of a Sonos, but search targets will vary by manufacturer and device. For UPnP, this information should be published on documentation for the device, but you may alternatively have to contact the manufacturer directly to obtain it. Here is the Event subscription:

```
subscribe(location, "ssdpTerm.urn:schemas-upnp-org:device:ZonePlayer:1", ssdpHandler)
```

This means that any time an SSDP **search response** with a search target of urn:schemas-upnp-org:device:ZonePlayer:1 (e.g. Sonos) is received from a Hub in this Location, it will fire the ssdpHandler method.

Next, you need to send an appropriate discovery command for the desired search target:

```
void ssdpDiscover() {
    sendHubCommand(new physicalgraph.device.HubAction("lan discovery urn:schemas-upnp-org:device:ZonePlayer:1"))
}
```

Note: HubAction is a class supplied by the SmartThings platform

The class `physicalgraph.device.HubAction` encapsulates request information for communicating with the device.

When you create an instance of a HubAction, you provide details about the request, such as the request method, headers, and path. By itself, HubAction is little more than a wrapper for these request details. In this case, it's a thin wrapper around discovery information.

In the above HubAction example, the main message to be sent through the Hub is:

```
lan_discovery_urn:schemas-upnp-org:device:ZonePlayer:1
```

This is converted by our device connectivity layer into an M-SEARCH multicast request that is sent to the LAN via the Hub, and should look something like the following:

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 4
ST: urn:schemas-upnp-org:device:ZonePlayer:1
```

After the end device receives the multicast M-SEARCH, it is supposed to issue a unicast **search response**, delayed by a random number of seconds between 0 and MX (4 in this case). The search response sent from the device back to the Hub should look something like this:

```
HTTP/1.1 200 OK
CACHE-CONTROL: max-age=100
EXT:
LOCATION: http://10.0.1.14:80/xml/device_description.xml
SERVER: FreeRTOS/6.0.5, UPnP/1.0, IpBridge/0.1
ST: urn:schemas-upnp-org:device:ZonePlayer:1
USN: uuid:RINCON_000E58F0FFFFFF400::urn:schemas-upnp-org:device:ZonePlayer:1
```

This will get routed back to the cloud where it will be converted into an Event that will fire the ssdpHandler method with the following description:

```
devicetype:04, mac:000E58F0FFFF, networkAddress:0A00010E, deviceAddress:0578, stringCount:04, ssdpPath:
```

The `ssdpHandler` method should record the data from the search response, in preparation for verification.

```
def ssdpHandler(evt) {
  def description = evt.description
  def hub = evt?.hubId

  def parsedEvent = parseEventMessage(description)
  parsedEvent << ["hub":hub]

  def devices = getDevices()
  String ssdpUSN = parsedEvent.ssdpUSN.toString()
  if (!devices."${ssdpUSN}") {
    devices << ["${ssdpUSN}": parsedEvent]
  }
}
```

146.2.2 Verification

Once we've recorded the presence of a device on the LAN with the desired SSDP search target, the next step is to verify the availability of the device by fetching some more information about it. In UPnP, this is called the **device description**. In the search response, there is a `LOCATION` header which shows the Location of the device description on the LAN. SmartThings splits this into **networkAddress**, **deviceAddress**, and **ssdpPath** in the Event, which at this point should exist in app state. This can be pulled out of state and put into a `HubAction`. Note that the `HubAction` has a **callback**, which means that when an HTTP response is issued from the device to the Hub, it will fire the **deviceDescriptionHandler** method.

```
void verifyDevices() {
  def devices = getDevices().findAll { it?.value?.verified != true }
  devices.each {
    int port = convertHexToInt(it.value.port)
    String ip = convertHexToIP(it.value.ip)
    String host = "${ip}:${port}"
    sendHubCommand(new physicalgraph.device.HubAction("""GET ${it.value.ssdpPath} HTTP/1.1\r\nHOST: ${host}"""))
  }
}

void deviceDescriptionHandler(physicalgraph.device.HubResponse hubResponse) {
  def body = hubResponse.xml
  def devices = getDevices()
  def device = devices.find { it?.key?.contains(body?.device?.UDN?.text()) }
  if (device) {
    device.value << [name: body?.device?.roomName?.text(), model: body?.device?.modelName?.text()]
  }
}
```

Note: `HubResponse` is a class supplied by the SmartThings platform. Here are some pieces of data that are included:

- **description** - The raw message received by the device connectivity layer
- **hubId** - The UUID of the SmartThings Hub that received the response
- **status** - HTTP status code of the response
- **headers** - Map of the HTTP headers of the response

- **body** - String of the HTTP response body
- **error** - Any error encountered during any automatic parsing of the body as either JSON or XML
- **json** - If the HTTP response has a Content-Type header of application/json, the body is automatically parsed as JSON and stored here
- **xml** - If the HTTP response has a Content-Type header of text/xml, the body is automatically parsed as XML and stored here

146.2.3 Inclusion

Now that the device has been verified, we need to add it as a child device.

```
def addDevices() {
    def devices = getDevices()

    selectedDevices.each { dni ->
        def selectedDevice = devices.find { it.value.mac == dni }
        def d
        if (selectedDevice) {
            d = getChildDevices()?.find {
                it.deviceNetworkId == selectedDevice.value.mac
            }
        }

        if (!d) {
            log.debug "Creating Generic UPnP Device with dni: ${selectedDevice.value.mac}"
            addChildDevice("smarththings", "Generic UPnP Device", selectedDevice.value.mac, selectedDevice.value.ip, selectedDevice.value.port,
                "label": selectedDevice?.value?.name ?: "Generic UPnP Device",
                "data": [
                    "mac": selectedDevice.value.mac,
                    "ip": selectedDevice.value.ip,
                    "port": selectedDevice.value.port
                ]
            )
        }
    }
}
```

Note: It's important to **not** use IP and port as the DNI (Device Network ID) of the device. This is because if/when the IP address changes, we do not want to update the device's DNI. Instead, we choose MAC address as DNI, which is guaranteed not to change.

146.2.4 Health

Lastly, we need to handle the possibility of IP address or port changes. Unless you have setup a static DHCP reservation in your network router, there is a possibility that the IP address of the device will change, and the child device can be told when this changes by the Service Manager. We'll start by modifying the above `ssdpHandler` method to handle changing IP and port data:

```
def ssdpHandler(evt) {
  def description = evt.description
  def hub = evt?.hubId

  def parsedEvent = parseEventMessage(description)
  parsedEvent << ["hub":hub]

  def devices = getDevices()
  String ssdpUSN = parsedEvent.ssdpUSN.toString()
  if (devices."${ssdpUSN}") {
    def d = devices."${ssdpUSN}"
    if (d.ip != parsedEvent.ip || d.port != parsedEvent.port) {
      d.ip = parsedEvent.ip
      d.port = parsedEvent.port
      def child = getChildDevice(parsedEvent.mac)
      if (child) {
        child.sync(parsedEvent.ip, parsedEvent.port)
      }
    }
  } else {
    devices << ["${ssdpUSN}": parsedEvent]
  }
}
```

This assumes that the DeviceType has a **sync** method that has the ability to alter the internally stored ip and port.

```
def sync(ip, port) {
  def existingIp = getDataValue("ip")
  def existingPort = getDataValue("port")
  if (ip && ip != existingIp) {
    updateDataValue("ip", ip)
  }
  if (port && port != existingPort) {
    updateDataValue("port", port)
  }
}
```

Finally, we need to make sure that the M-SEARCH for our desired search target is periodically sent out over the LAN. We can use the scheduler to do that from the Service Manager:

```
runEvery5Minutes("ssdpDiscover")
```

146.2.5 Best practices

For LAN Service Manager SmartApps, there are a couple items to keep in mind that might not be immediately apparent.

- Use something static as the DNI for the child device, such as MAC address.
 - Avoid making calls from your child devices into the parent if possible, as this can lead to increased latency and unnecessary platform load. Instead, supply your child devices with enough information to make calls into the parent unnecessary, and use the Service Manager to manage any child device updates that need to happen based on network changes.
-

146.2.6 References and resources

- UPnP Device Architecture 1.1

146.3 Building the Device Type

The Device Handler for a LAN-connected device is generally the same as any other Device Handler. The means in which it handles sending and receiving messages from its device is a little bit different. Let's walk through a LAN-connected Device Handler example.

146.3.1 Making outbound HTTP calls with HubAction

Depending on the type of device you are using, you will send requests to your devices through the Hub via REST or UPnP. You can do this using the SmartThings provided `HubAction` class.

146.3.2 Overview

The class `physicalgraph.device.HubAction` encapsulates request information for communicating with the device.

When you create an instance of a `HubAction`, you provide details about the request, such as the request method, headers, and path. By itself, `HubAction` is little more than a wrapper for these request details.

It is when an instance of a `HubAction` is returned from a command method that it becomes useful.

When a command method of your Device Handler returns an instance of a `HubAction`, the SmartThings platform will use the request information within it to actually perform the request. It will then call the device-handler's `parse` method with any response data.

Herein lies an important point - *if your `HubAction` instance is not returned from your command method, no request will be made.* It will just be an object allocating system memory. Not very useful.

So remember - the `HubAction` instance should be returned from your command method so that the platform can make the request!

146.3.3 Creating a HubAction object

To create a `HubAction` object, you can pass in a map of parameters to the constructor that defines the request information:

```
def result = new physicalgraph.device.HubAction(  
    method: "GET",  
    path: "/somepath",  
    headers: [  
        HOST: "device IP address"  
    ],  
    query: [param1: "value1", param2: "value2"]  
)
```

A brief discussion of the options that can be provided follows:

method The HTTP method to use for the request.

path The path to send the request to. You can add URL parameters to the request directly, or use the `query` option.

headers A map of HTTP headers and their values for this request. This is where you will provide the IP address of the device as the `HOST`.

query A map of query parameters to use in this request. You can use URL parameters directly on the path if you wish, instead of using this option.

146.3.4 Parsing the response

When you make a request to your device using `HubAction`, any response will be passed to your device-handler's `parse` method, just like other device messages.

You can use the `parseLanMessage` method to parse the incoming message.

`parseLanMessage` example:

```
def parse(description) {
    ...
    def msg = parseLanMessage(description)

    def headersAsString = msg.header // => headers as a string
    def headerMap = msg.headers // => headers as a Map
    def body = msg.body // => request body as a string
    def status = msg.status // => http status code of the response
    def json = msg.json // => any JSON included in response body, as a data structure of
    def xml = msg.xml // => any XML included in response body, as a document tree structure
    def data = msg.data // => either JSON or XML in response body (whichever is specified)

    ...
}
```

For more information about the JSON or XML response formats, see the Groovy [JsonSlurper](#) and [XmlSlurper](#) documentation.

146.3.5 Getting the addresses

To use `HubAction`, you will need the IP address of the device, and sometimes the Hub.

How the device IP and port are stored may vary depending on the device type. There's currently not a public API to get this information easily, so until there is, you will need to handle this in your device-type handler. Consider using helper methods like these to get this information:

```
// gets the address of the Hub
private getCallbackAddress() {
    return device.hub.getDataValue("localIP") + ":" + device.hub.getDataValue("localSrvPortTCP")
}

// gets the address of the device
private getHostAddress() {
    def ip = getDataValue("ip")
    def port = getDataValue("port")
}
```

```

if (!ip || !port) {
    def parts = device.deviceNetworkId.split(":")
    if (parts.length == 2) {
        ip = parts[0]
        port = parts[1]
    } else {
        log.warn "Can't figure out ip and port for device: ${device.id}"
    }
}

log.debug "Using IP: $ip and port: $port for device: ${device.id}"
return convertHexToIP(ip) + ":" + convertHexToInt(port)
}

private Integer convertHexToInt(hex) {
    return Integer.parseInt(hex, 16)
}

private String convertHexToIP(hex) {
    return [convertHexToInt(hex[0..1]), convertHexToInt(hex[2..3]), convertHexToInt(hex[4..5]), convertHexToInt(hex[6..7])]
}

```

You'll see the rest of the examples in this document use these helper methods.

146.3.6 Wake on LAN (WOL)

HubAction can be used to make WOL requests.

Here is an example:

```

def myWOLCommand() {
    def result = new physicalgraph.device.HubAction (
        "wake on lan <your mac address w/o ':'>",
        physicalgraph.device.Protocol.LAN,
        null,
        [secureCode: "111122223333"]
    )
    return result
}

```

The first argument to HubAction tells the HubAction class that this will be a WOL request. The argument must be in the form “wake on lan <mac address>” where the mac address is the address without the ‘:’ separator characters. For example, if the mac address of the NIC is 01:23:45:67:89:ab, the first parameter to HubAction would be “wake on lan 0123456789ab”.

The second parameter simply specifies that the request will be a LAN request. This will always be the case for a WOL type request. So the value must always be physicalgraph.device.Protocol.LAN.

The third parameter is the Device Network ID, or dni. In the case of a WOL request, this parameter should be null.

The last parameter is a map representing the options on the request. For a WOL request, this map will only ever consist of one parameter, secureCode. Some NIC's support the *SecureOn* feature which requires the request to not only have a valid mac address, but also supply a valid password. This password must be configured on the NIC. If the NIC does not support *SecureOn* or does not have a password set, simply leave out the options map.

146.3.7 REST requests

HubAction can be used to make REST calls to communicate with the device.

Here's a quick example:

```
def myCommand() {
    def result = new physicalgraph.device.HubAction(
        method: "GET",
        path: "/yourpath?param1=value1&param2=value2",
        headers: [
            HOST: getHostAddress()
        ]
    )
    return result
}
```

146.3.8 UPnP/SOAP requests

Alternatively, after making the initial connection you can use UPnP. UPnP uses SOAP (Simple Object Access Protocol) messages to communicate with the device.

SmartThings provides the HubSoapAction class for this purpose. It is similar to the HubAction class (it actually extends the HubAction class), but it will handle creating the soap envelope for you.

Here's an example of using HubSoapAction:

```
def someCommandMethod() {
    return doAction("SetVolume", "RenderingControl", "/MediaRenderer/RenderingControl/Control", [Inst
}]

def doAction(action, service, path, Map body = [InstanceID:0, Speed:1]) {
    def result = new physicalgraph.device.HubSoapAction(
        path: path,
        urn: "urn:schemas-upnp-org:service:$service:1",
        action: action,
        body: body,
        headers: [Host:getHostAddress(), CONNECTION: "close"]
    )
    return result
}
```

146.3.9 Subscribing to device Events

If you'd like to hear back from a LAN-connected device upon a particular Event, you can subscribe using a HubAction. The parse method will be called when this Event is fired on the device.

Here's an example using UPnP:

```
def someCommand() {
    subscribeAction("/path/of/event")
}

private subscribeAction(path, callbackPatl="") {
```

```
log.trace "subscribe($path, $callbackPath)"
def address = getCallbackAddress()
def ip = getHostAddress()

def result = new physicalgraph.device.HubAction(
    method: "SUBSCRIBE",
    path: path,
    headers: [
        HOST: ip,
        CALLBACK: "<http://${address}/notify$callbackPath>",
        NT: "upnp:event",
        TIMEOUT: "Second-28800"
    ]
)

log.trace "SUBSCRIBE $path"

return result
}
```

146.3.10 References and resources

- UPnP
- SOAP
- REST

Automatic LAN Device Discovery

Automatic LAN device discovery minimizes the complexity in discovering LAN-connected devices.

Normally the SmartThings platform will discover a LAN-connected or a Cloud-connected device only when a Service Manager SmartApp for that specific device is present. This means that if you want to integrate multiple LAN devices, such as a Wemo motion sensor and a Bose Speaker, then you will need multiple Service Manager SmartApps, i.e., a separate Service Manager SmartApp for each LAN-connected device. On the contrary, the platform does not have any such Service Manager SmartApp requirement for a ZigBee or a Z-Wave device.

The new automatic LAN device discovery eliminates the Service Manager SmartApp requirement for some LAN-connected devices, thereby making for a much smoother and quicker LAN-connected device discovery. See *Supported LAN-connected Devices* (page 573).

147.1 Impact on the developer

For the *Supported LAN-connected Devices* (page 573) if you have made any customizations to either your Service Manager SmartApp or your Device Handler, then your LAN-connected device integration will be impacted. See the table below.

Custom Device Handler	Custom Service Manager SmartApp	Impact
Yes	No	Custom LAN Device Handler is overwritten with the SmartThings version.
Yes	Yes	No impact

147.1.1 Supported LAN-connected Devices

Currently a limited number of LAN-connected devices can be discovered with automatic LAN device discovery. See *How to connect Wi-Fi devices*.

Capturing and Displaying Camera Pictures

Cameras connected to SmartThings can use the imageCapture Capability, along with the Carousel Tile, to capture and view images. SmartThings-connected cameras are either LAN- or Cloud-Connected; this document outlines the steps to capture and display images for both.

148.1 Image Capture Capability

Add support for the imageCapture Capability by including it in the Device Handler's metadata:

```
metadata {
    definition(name: "My Camera Device", namespace: "MyNamespace", author: "My Name") {
        capability "Image Capture"
        // other definition metadata...
    }
}
```

The Image Capture Capability defines one attribute, “image”, and one command, `take()`. The Carousel Tile can be used to display images and allow the user to manually take a photo, as discussed next.

148.2 Tiles for taking and viewing pictures

Add tiles to allow the viewing and taking of images:

```
tiles {
    standardTile("image", "device.image", width: 1, height: 1, canChangeIcon: false, inactiveLabel: t
        state "default", label: "", action: "", icon: "st.camera.dropcam-centered", backgroundColor:
    }

    carouselTile("cameraDetails", "device.image", width: 3, height: 2) { }

    standardTile("take", "device.image", width: 1, height: 1, canChangeIcon: false, inactiveLabel: t
        state "take", label: "Take", action: "Image Capture.take", icon: "st.camera.dropcam", backgro
        state "taking", label: 'Taking', action: "", icon: "st.camera.dropcam", backgroundColor: "#00
        state "image", label: "Take", action: "Image Capture.take", icon: "st.camera.dropcam", backg
    }
}
```

```
main "image"
  details(["cameraDetails", "take"])
}
```

The `carouselTile` is where the images will be displayed, and the “take” tile allows users to capture images. Note that both are associated with the `"image"` attribute; this association allows the images to be taken and displayed properly.

148.3 Capture and display images

The `take()` command of the Image Capture Capability is responsible for capturing the image. Follow the protocol-specific instructions for implementing this command method below.

148.3.1 LAN-connected cameras

LAN-connected devices can capture images using *HubAction* (page 1029), store them using *storeTemporaryImage()* (page 977), and display them with the *Carousel Tile* (page 486).

The `take()` command will issue a request to take a picture via a *HubAction*. The response from the device will be sent to the Device Handler’s `parse()` method, where it can then be moved to longer-lasting storage using `storeTemporaryImage()`. `storeTemporaryImage()` also emits the “image” event, causing the *Carousel Tile* to be updated with the new image.

Here’s an example of the `take()` command (details of the request will be specific to each device):

```
def take() {
  def host = getHostAddress()
  def port = host.split(":")[1]

  def path = "/some/path/"

  def hubAction = new physicalgraph.device.HubAction(
    method: "GET",
    path: path,
    headers: [HOST:host]
  )

  hubAction.options = [outputMsgToS3:true]

  return hubAction
}

/**
 * Utility method to get the host addresses
 */
private getHostAddress() {
  def parts = device.deviceNetworkId.split(":")
  def ip = convertHexToIP(parts[0])
  def port = convertHexToInt(parts[1])
  return ip + ":" + port
}
```

Some things to note about the implementation of the `take()` command:

1. The specific path, method, and headers of the *HubAction* will vary for each device. Consult the device manufacturer’s documentation for this information.

2. Make sure to specify `hubAction.options = [outputMsgToS3: true]`. This will result in the image being stored (temporarily). We will move the image to longer-lasting storage next.
3. Remember to return the `HubAction` from the command method, otherwise it will not be executed!

Once we've made the request in the `take()` command method, the response from the device will be sent to the Device Handler's `parse()` method. This response will contain a `tempImageKey`, which is the key of the photo just taken.

```
def parse(String description) {
    def map = stringToMap(description)

    if (map.tempImageKey) {
        try {
            storeTemporaryImage(map.tempImageKey, getPictureName())
        } catch (Exception e) {
            log.error e
        }
    } else if (map.error) {
        log.error "Error: ${map.error}"
    }

    // parse other messages too
}

private getPictureName() {
    return java.util.UUID.randomUUID().toString().replaceAll('-', '')
}
```

`parse()` does the following:

1. Checks the response to see if `tempImageKey` was sent. If it was, this means that this is the image response from our `take()` command.
2. Calls `storeTemporaryImage()` with the `tempImageKey` and a name for the picture. The name must be unique per device instance, contain only alphanumeric, “-”, “_”, and “.” characters. This will move the image from temporary storage to a location where the image will be stored for 365 days, before being permanently deleted.

`storeTemporaryImage()` also emits the “image” event, which is the attribute our Carousel Tile is associated with. This is what allows the image to be displayed in the tile.

148.3.2 Cloud-connected cameras

The `take()` command will issue an HTTP request to the third-party service to capture the image, and store the resulting image bytes using `storeImage()` (page 976).

Below is a simplified example (A real application will need to handle authentication with the third-party, as well as additional error handling):

```
def take() {
    def params = [
        uri: "https://some-uri",
        path: "/some/path"
    ]

    try {
        httpGet(params) { response ->
```

```
// we expect a content type of "image/jpeg" from the third party in this case
if (response.status == 200 && response.headers.'Content-Type'.contains("image/jpeg")) {
    def imageBytes = response.data
    if (imageBytes) {
        def name = getImageName()
        try {
            storeImage(name, imageBytes)
        } catch (e) {
            log.error "Error storing image ${name}: ${e}"
        }
    }
} else {
    log.error "Image response not successful or not a jpeg response"
}
} catch (err) {
    log.debug "Error making request: $err"
}
}

def getImageName() {
    return java.util.UUID.randomUUID().toString().replaceAll('-', '')
}
}
```

Warning: Only synchronous HTTP requests are supported when using the Carousel Tile.

The `take()` command above does the following:

1. Makes a request to a URI that will return an image response. A real integration would need to provide authorization information on the request. This would typically be an OAuth token obtained through the setup process, as documented [here](#) (page 546).
2. If the response is successful and its `Content-Type` is our expected content, it gets the image bytes from `response.data`.
3. Stores the image using `storeImage()`, using a name generated from a UUID. The name of the image is required to be unique for each device instance.

`storeImage()` will emit the “image” event, which causes the Carousel Tile to be updated with the new image.

Tip: `httpClient.get()` will serialize the response data for images into a `ByteArrayInputStream`, which is why we can pass the response body to `storeImage()`.

148.4 Retrieving an image

If you need to retrieve the byte representation of an image stored with `storeImage()` or `storeTemporaryImage()`, use `getImage()` (page 959). This will return the bytes of the image in a `ByteArrayInputStream`.

```
// Image with "some-name" that was previously stored
ByteArrayInputStream img = getImage("some-name")
```

148.5 Image size limits

Images are limited to a maximum of one megabyte.

`storeImage()` will throw an `InvalidParameterException` if this limit is exceeded.

Attempting to capture an image exceeding this maximum using `HubAction` will result in the message sent to `parse()` containing an error response:

```
def parse(String description) {
    def map = stringToMap(description)

    if (map.error) {
        log.error "error: ${map.error}"
    } else if (map.tempImageKey) {
        //...
    }
}
```

148.6 Allowed image name characters

Image names are restricted to alphanumeric, "-", "_", and "." characters.

An `InvalidParameterException` is thrown by `storeTemporaryImage()` and `storeImage()` if the name contains other characters.

148.7 Image storage duration

Images stored via a `HubAction` are stored for 24 hours, after which it is deleted (this is why we use `storeTemporaryImage()` to move images captured by a `HubAction`).

Images stored via `storeImage()` or `storeTemporaryImage()` are available to clients for seven days, and stored by SmartThings for 365 days, after which it is deleted.

148.8 Supported image formats

`storeImage()` supports both JPEG and PNG image formats. The content type can be specified when calling `storeImage()`:

```
storeImage("some-image-name", imgBytes, "image/png")
```

The content type of "image/jpeg" is the default.

Images captured via a `HubAction` and stored with `storeTemporaryImage()` must be in JPEG format.

In either case, there is no need to include the file extension (e.g., ".jpg" or ".png" in the image name).

148.9 Related documentation

- [storeTemporaryImage\(\) reference documentation](#) (page 977)
- [storeImage\(\) reference documentation](#) (page 976)
- [HubAction reference documentation](#) (page 1029)
- [Image Capture Capability reference documentation](#)
- [Tiles documentation](#) (page 473)

Part XIV

Composite Devices

Devices such as Hue LAN bridge, AEON Z-Wave SmartStrip, or a Zooz ZEN20 Z-Wave Power Strip have multiple components, and each component can be controlled independently. For example, a Zooz ZEN20 Z-Wave Power Strip can be used with a separate Thing connected to each of its five outlets and each Thing can have its own SmartApp.

SmartThings categorizes such a multiple-component device as a *composite device*. A device is said to be a composite device when it treats each of its component as its child device. Integrating a composite device into SmartThings platform involves incorporating the composite device functionality into its Device Handler. Additionally, you may need to modify the Service Manager SmartApp and the SmartApp.

Device Handler for a Composite Device

When you integrate a composite device into SmartThings, the composite device maintains a parent-child relationship between itself and its child devices. For example, the Device Handler of Zooz ZEN20 Z-Wave Power Strip composite device implements the Power Strip as a parent device and each outlet as a separate child device. More specifically, each individual outlet of the Power Strip is implemented as a *child device instance* of Zooz Power Strip Outlet, whereas the Power Strip itself is an instance of Zooz Power Strip as a *parent device*.

Similarly, the Hue bridge Device Handler implements the Hue bridge as a parent device and Hue bulbs as child devices of the Hue bridge parent device.

149.1 Parent Device Handler

Let's look at how to set up a parent Device Handler. For example, in the Device Handler of the Zooz ZEN20 Z-Wave Power Strip composite device, the parent device functionality shown below:

- Creates a *child device instance* of Zooz Power Strip Outlet device for each outlet of the Power Strip, by using the `addChildDevice()` (page 949) method, as below:

```

metadata {
    definition (name: "ZooZ Power Strip", namespace: "smarththings", author: "SmartThings") {
        capability "Switch"
        capability "Refresh"
        capability "Configuration"
        capability "Actuator"
        capability "Sensor"
        fingerprint deviceId: "0x1004", inClusters: "0x5E,0x85,0x59,0x5A,0x72,0x60,0x8E,0x73,0x27,0x28"
    }
}

...

def installed() {
    createChildDevices ()
    response (refresh () + configure ())
}

...

private void createChildDevices () {
    // Save the device label for updates by updated()
    state.oldLabel = device.label
    // Add child devices for all five outlets of Zooz Power Strip
    for (i in 1..5) {

```

```
        addChildDevice("ZooZ Power Strip Outlet", "${device.deviceNetworkId}-${i}", null, [completedS  
    }  
}
```

and,

- Creates child device APIs such as:

```
void childOn(String dni) {  
    onOffCmd(0xFF, channelNumber(dni))  
}  
void childOff(String dni) {  
    onOffCmd(0, channelNumber(dni))  
}
```

149.2 Child Device Handler

Next, the below Device Handler code sets up the *outlet* of the Zooz ZEN20 Z-Wave Power Strip device as the *child device instance*.

```
metadata {  
definition (name: "ZooZ Power Strip Outlet", namespace: "smarththings", author: "SmartThings") {  
    capability "Switch"  
    capability "Actuator"  
    capability "Sensor"  
}  
...  
void on() {  
    parent.childOn(device.deviceNetworkId)  
}  
void off() {  
    parent.childOff(device.deviceNetworkId)  
}
```

In the above example, the method calls, `parent.childOn(device.deviceNetworkId)` and `parent.childOff(device.deviceNetworkId)`, are the means of communication between the parent and the child instances of this composite device.

Deleting a Composite Device

Deleting a composite parent device will delete all children devices. For example, deleting the Power Strip itself will delete its outlets as devices from the SmartThings platform.

SmartApps can be configured to control individual outlets as well as the entire power strip. In such a case, if you try to delete the Power Strip parent device itself, then you are given an option to force-delete the outlet device.

If you try to delete a composite device from your SmartThings mobile app, then the following applies:

- If the parameter `isComponent` is set to `true`, as shown in the *Parent Device Handler* (page 585) example above, then the device is hidden from the Things view and you will not be presented with the option of deleting child devices individually.
- If the parameter `isComponent` is set to `false`, then you can delete individual child devices.

Note: Note that the following applies for a composite device:

- A single SmartApp can control all the components, each independently, sending and receive messages from each component device.
 - A single SmartApp can control all components together in an all-or-nothing fashion.
-

Composite Device Tiles

Child device tiles can be visually pulled together into a composite tile. On SmartThings mobile app, such a composite tile represents a rich interface for the display and control of a composite device.

For example, consider a refrigerator composite device that is built with two child components, i.e., the fridge door and the temperature control.

In the fridge door child Device Handler, the tile for the fridge door `mainDoor` is defined normally with the `standardTile` method, as below:

```
// Fridge door child component Device Handler
metadata {
    definition (name: "Simulated Refrigerator Door", namespace: "smarththings/testing", author: "SmartThings") {
        capability "Contact Sensor"
        capability "Sensor"
        capability "open"
        capability "close"
    }
    tiles {
        standardTile("mainDoor", "device.contact", width: 2, height: 2, decoration: "flat") {
            state("closed", label: 'Fridge', icon: "st.contact.contact.closed", backgroundColor: "#79b821")
            state("open", label: 'Fridge', icon: "st.contact.contact.open", backgroundColor: "#ffa81e")
        }
    }
    ...
}
```

Then, by using the method `childDeviceTile()` (page 953) within the refrigerator parent Device Handler, we can customize how the above fridge door tile `mainDoor` is pulled visually into the refrigerator composite tile. See below:

```
// Refrigerator parent Device Handler
metadata {
    definition (name: "Simulated Refrigerator", namespace: "smarththings/testing", author: "SmartThings") {
        capability "Contact Sensor"
    }
    tiles {
        childDeviceTile("mainDoor", "mainDoor", height: 2, width: 2, childTileName: "mainDoor")
    }
    ...
}
```

The example below illustrates how to put together a mobile visual interface on SmartThings mobile app for a simulated refrigerator composite device.

151.1 Example: Simulated refrigerator

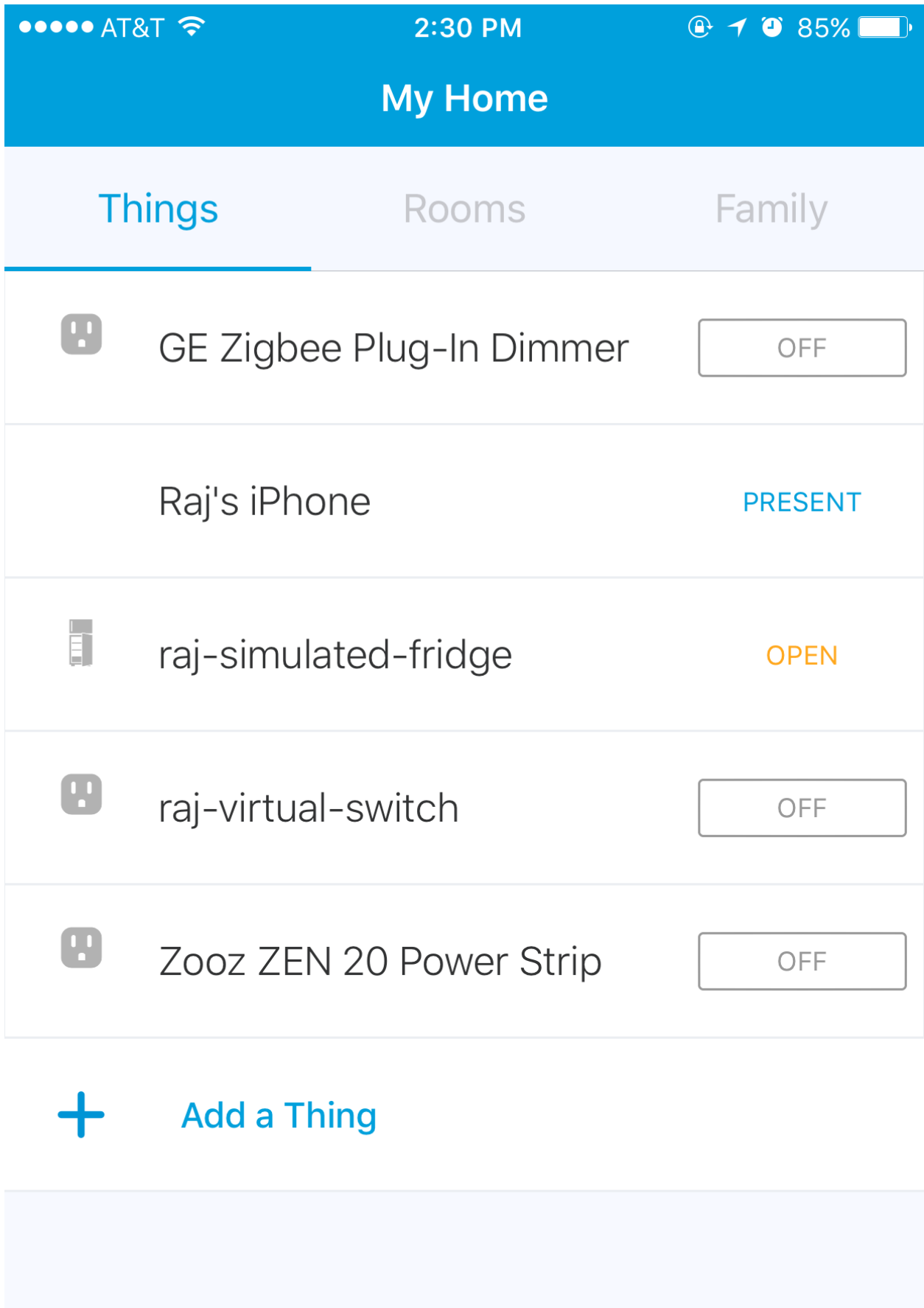
The simulated refrigerator in this example is a composite device with two components (child devices):

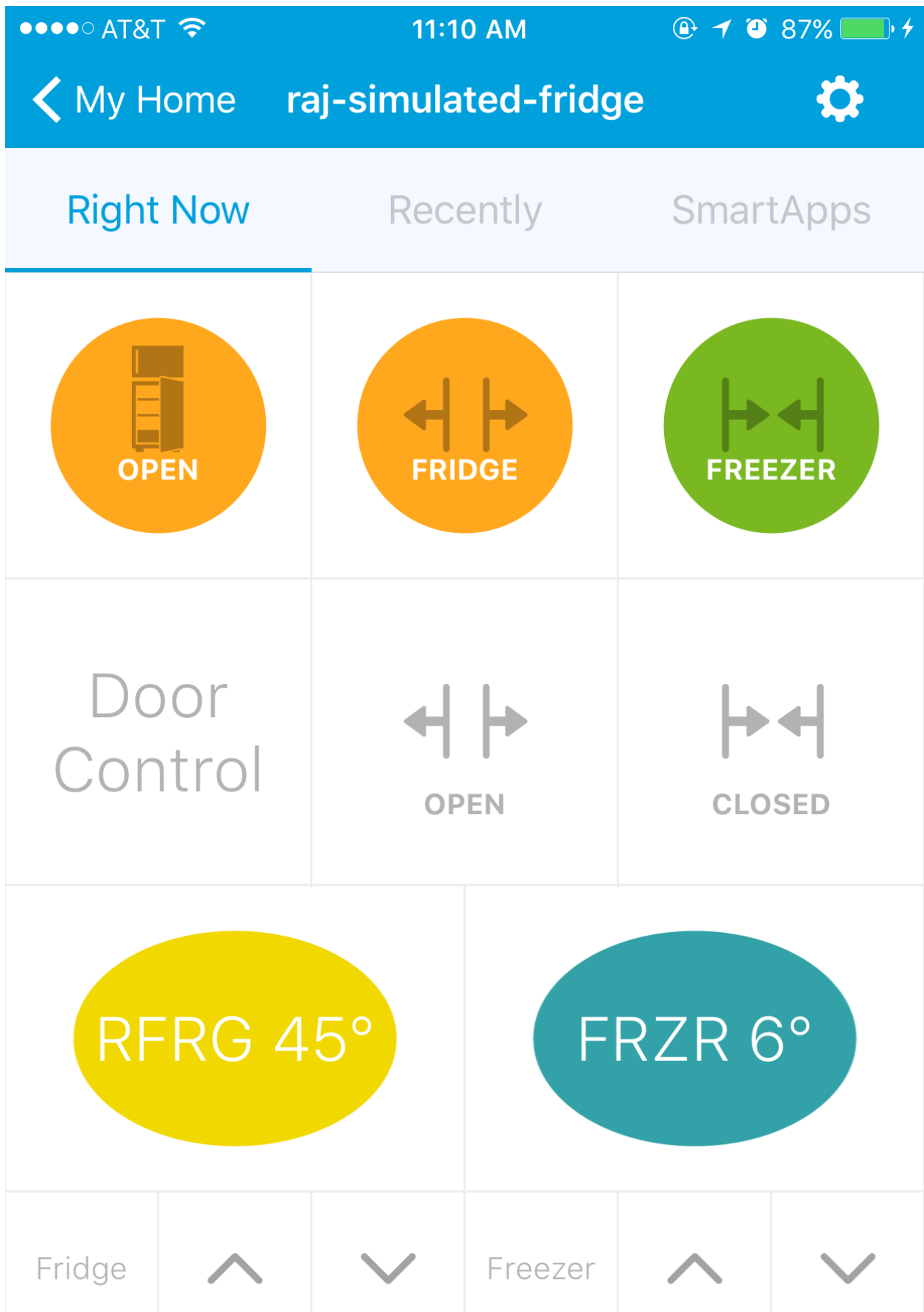
- The simulated main refrigerator (fridge) compartment, and
- A simulated freezer compartment.

Each compartment has its own door, its own temperature, and its own temperature setpoint. Each compartment is modeled as a child device of the main refrigerator device.

From IDE, create a *New Device* (see *Create a Virtual Device* (page 453)) and set it to *Type* “Simulated Refrigerator”. This will create the composite parent device *Simulated Refrigerator*. You will see it appear in the *Things* view of your SmartThings mobile app. Tap on it to see the *Detail* view of it.

The mobile app view of the Simulated Refrigerator composite device, with the detail view on the right, looks as below:





Note: If you are new to SmartThings tiles, see [Tiles](#) (page 473) before you proceed further.

The composite device tile for the refrigerator door, shown in the top row of the detail view above, is put together as below:

- In the child Device Handler for the Simulated Refrigerator Door, the tile `mainDoor` is defined in the `tiles()` section. The width and height parameters defined here will be overridden by the parent Device Handler setting.

```

metadata {
    definition (name: "Simulated Refrigerator Door", namespace: "smarththings/testing", author: "SmartThings") {
        capability "Contact Sensor"
        capability "Sensor"
        command "open"
        command "close"
    }
    tiles {
        standardTile("mainDoor", "device.contact", width: 2, height: 2, decoration: "flat") {
            state("closed", label: 'Fridge', icon: "st.contact.contact.closed", backgroundColor: "#79b82c")
            state("open", label: 'Fridge', icon: "st.contact.contact.open", backgroundColor: "#ffa81e")
        }
    }
}

```

- In the Simulated Refrigerator parent Device Handler, the method `childDeviceTile()` (page 953) is used in the `tiles()` section to visually configure this child device `mainDoor` tile. The width and height settings here will override the settings for this tile in the child Device Handler.

```

metadata {
    definition (name: "Simulated Refrigerator", namespace: "smarththings/testing", author: "SmartThings") {
        capability "Contact Sensor"
    }
    tiles {
        childDeviceTile("mainDoor", "mainDoor", height: 2, width: 2, childTileName: "mainDoor")
    }
    ...
}
def installed() {
    state.counter = state.counter ? state.counter + 1 : 1
    if (state.counter == 1) {
        // A tile with the name "mainDoor" exists in the tiles() method of the child Device Handler
        addChildDevice(
            "Simulated Refrigerator Door",
            "${device.deviceNetworkId}.2",
            null,
            [completedSetup: true, label: "${device.label} (Main Door)", componentName: "mainDoor",

```

Note: While the width and height parameters in the `childDeviceTile()` in the parent Device Handler will override the settings of these parameters in the child Device Handler, any icon setting specified in the child Device Handler will *not* be overridden by the `childDeviceTile()`.

151.2 Example composite tile code

Copy the following three composite device Device Handler files and create your own three Device Handlers with *From Code* option (see *Create a new Device Handler* (page 451)):

- Parent Device Handler file for the [Simulated Refrigerator](#) composite parent device.
- Child Device Handler file for the [Simulated Refrigerator Door](#) component device, and
- Child Device Handler for the [Simulated Refrigerator Temperature Control](#) component device.

Note: Make sure to publish *For Me* the above three Device Handlers before you proceed further.

Follow the code in the Device Handlers you copied over to see how the rest of the visual layout is configured for the entire Simulated Refrigerator composite device.

Part XV

Arduino ThingShield

Warning: The SmartThings Arduino ThingShield has been discontinued, and is no longer supported. All code and libraries discussed in this document are no longer supported by SmartThings, and should be used on a as-is basis.

Using the SmartThings Arduino Shield (ThingShield), you can add SmartThings capability to any Arduino compatible board with the R3 pinout, including the Uno, Mega, Duemilanove, and Leonardo.

Specs:

- Works with: Uno, Mega, Duemilanove, Leonardo
- Dimensions: 2.5 x 1.9 x 0.3"
- Weight: 8 ounces

Installing the library

To install, copy the entire SmartThings directory into the 'libraries' directory in your sketchbook. Your sketchbook location is set in the Arduino IDE preferences, by default, the location will be:

Windows: 'My Documents\Arduino\libraries\SmartThings'

OSX: '~/Documents/Arduino/libraries/SmartThings'

You can download the [SmartThings Arduino Library](#) here.

Pairing the shield

To join the shield to your SmartThings Hub, go to “Add SmartThings” mode in the SmartThings app by hitting the “+” icon in the desired location, and then press the Switch button on the shield. You should see the shield appear in the app.

To unpair the shield, press and hold the Switch button for 6 seconds and release. The shield will now be unpaired from your SmartThings Hub. Make sure to delete from your account if you plan to re-pair it!

Changing the Device Handler

By changing the Device Handler in the SmartThings cloud you can change how to interact with your Arduino + ThingShield. When a shield first pairs, it has no functionality and only serves to help identify the device in the mobile app. We have some pre-built Device Handlers that you can use for most functionality. One pre-built Arduino Device Handler is the “On/Off Shield (example)”

To change your Device Handler, log into <http://graph.api.smarthings.com/> and click on “Devices” Navigate to and click on the Arduino ThingShield then click on “Edit” on the bottom left of the page.

Select the “Type” drop down menu.

Choose “On/Off Shield (example)”

Hit the “Update” button

Your Arduino will now be able to accept the commands “on” “off”, and “hello”

[Here is what the Arduino sketch looks like](#) and [here is the Device Handler](#).

[Here is a different Device Handler that can read a string sent from an Arduino and display it in a tile.](#)

Arduino examples

We have created some example Arduino Sketches (code) to use as a reference for building your own devices. The following is meant to go with the "On/Off Shield (example)" Device Handler.

[Download all of our examples here.](#)

Part XVI

Rate Limits

Rate limiting ensures that no single SmartApp or Device Handler will consume too many shared resources. Rate limits apply to **all** SmartApps and Device Handlers.

SmartApp and Device Handler rate limits

SmartApps and Device Handlers are monitored for excessive resource utilization on two measures: *Execution count limits* and *Execution time limits*.

156.1 Execution count limits

SmartApps and Device Handlers are subject to the following execution count limits. These limits are per *installed SmartApp or Device Handler*.

Execution count limit	Time window	Description
250 executions	60 seconds	A maximum of 250 executions per minute is allowed for each installed SmartApp or Device Handler.

If the limit is exceeded, an error will be displayed in Live Logging, and no further executions for this installed SmartApp or Device Handler will occur until the current 60-second time window expires.

156.2 Execution time limits

These execution time limits apply to SmartApps and Device Handlers:

What	Limit
Method execution time	20 seconds
Total continuous execution time	40 seconds

If these limits are exceeded, the current execution will be suspended.

Web services rate limit headers

SmartApps and Device Handlers that expose RESTful APIs are subject to the same rate limits as documented above. The SmartThings platform will set three HTTP headers on the response for every inbound API call, so that a client may understand the current rate limit status.

Header	Description
X-RateLimit-Limit	The total enforced rate limit (250)
X-RateLimit-Curr	The number of executions within the current rate limit time window, for this installed SmartApp or Device Handler.
X-RateLimit-TTL	The time remaining (in seconds) before the current rate limit window resets, for this installed SmartApp or Device Handler.

If the rate limit is exceeded, the following response is sent to the client:

HTTP Response Code	Error Response
429 (Too Many Requests)	<code>{"error": true, "type": "RateLimit", "message": "Please try again later"}</code>

SMS rate limits

The following limits apply to sending SMS messages:

Limit	If exceeded
15 SMS messages per number, per 60 seconds	No additional SMS messages will be sent until the next minute.

Note: This limit applies **per number**, not per SmartApp or user.

Parent-child relationship limit

The number of child SmartApps or child devices that a SmartApp or Device Handler may have are subject to the following limits:

Maximum child count	Description
500	A SmartApp may have at most a combination of 500 child SmartApps or Devices. A Device Handler may have at most 500 child Devices.

If this limit is exceeded, an exception is thrown and will be displayed in Live Logging. If initiated from within the mobile app, an error will be seen in the mobile application as well.

Avoiding rate limits

While SmartThings rate limits are quite high compared to other service platforms, the event-driven nature of SmartThings can result in SmartApps or Device Handlers that may (unintentionally) reach this limit. It is important to reason carefully about your code, think of worst-case scenarios, and monitor Live Logging when testing to reduce the likelihood of being rate limited.

Here are some common pitfalls to watch out for:

- A SmartApp may subscribe to a large number of “chatty” devices, causing the execution limit to be reached. For example, DLNA devices may be particularly chatty, and frequently changing energy/power values may cause the rate limit to be exceeded.
- Service Manager SmartApps that may be called by their child devices may reach the execution limit, if there are a number of child devices and/or they call the parent in response to frequent events.
- Synchronous (blocking) HTTP requests may hit the execution time rate limit, depending on the third party response time. Avoid this possibility by using *Making Asynchronous External HTTP Requests (Beta)* (page 371).
- It’s possible to create an infinite loop of events. For example, subscribing to both “on” and “off” events, and the “on” command triggers the “off” event and vice versa - leading to a never-ending chain of event handlers being called.
- Pay attention to any looping logic around creating child devices or SmartApps. Any error in the looping logic might result in creating too many children, which could encounter the parent-child limit.

Part XVII

Publishing Code

You can publish your SmartApp or Device Handler either just for yourself, or for public distribution. Publishing for distribution requires you to submit your SmartApp or Device Handler for review and approval by SmartThings.

For yourself

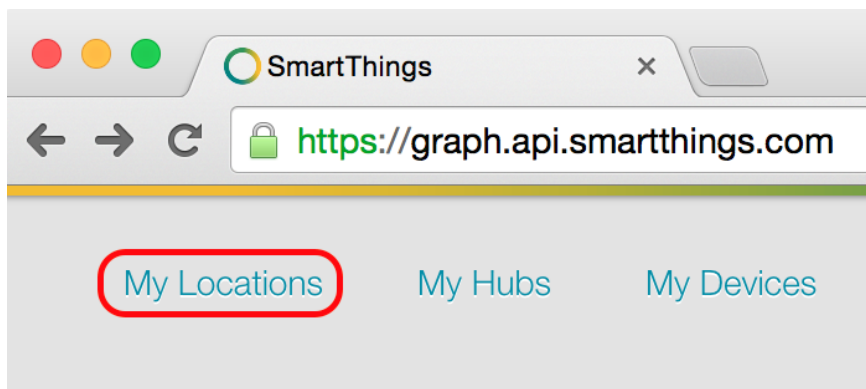
When you publish for yourself, your SmartApp or Device Handler is available *only to your account* on your SmartThings mobile app.

161.1 Ensure proper Location

To publish your SmartApp or Device Handler properly, you must be at the proper Location *before* you begin writing your code.

To ensure you are at the correct Location, follow these steps:

- Start by logging into IDE at <https://graph.api.smarthings.com>.
- Next, navigate to *My Locations* page.



- On this page is a listing of all the Locations you created. Normally you will see just one Location where you installed your Hub.
- Click on the Location name appearing in the far left column (i.e., the *Name* column). You may need to log in again with your SmartThings userid and password.

Note: Note that even though the IDE is located at <https://graph.api.smarthings.com>, it may not always be the correct URL for your SmartApp or Device Handler deployment. By explicitly selecting the Location name you will ensure that your SmartApp or Device Handler will be published properly.

161.2 Publish

Next, to publish for yourself, follow these steps:

- Make sure that you are in the proper Location (see above).
- From your SmartApp or Device Handler view, click on *Publish* button and click the *For Me* option.

This will publish your SmartApp or Device Handler for only your account. Open your SmartThing mobile app, navigate to *Marketplace* and choose *SmartApps* section. Tap on the *My Apps* category at the bottom and you will see your SmartApp.

For public distribution

Note: SmartThings is not reviewing submissions for public distribution at this time.

To publish your SmartApp or Device Handler for public distribution, you will need to submit it for review and approval by SmartThings. Follow these steps:

- On IDE, click on *My Publication Requests* in the top navigation bar. This will take you to your *Publication Requests* page.
- From this page click on *+New Request*. This will take you to *Submit a SmartApp or device type for publication* page.
- Follow the instructions on this page to submit your SmartApp or Device Handler for review by SmartThings.

162.1 Review process

SmartThings team will review your SmartApp or Device Handler for approval.

Note: To enhance the chances of your code getting your SmartApp or Device Handler approved, review and ensure your code follows the *Code Review Guidelines and Best Practices* (page 631).

Your SmartApp will be reviewed for the following criteria:

- Does this SmartApp duplicate an existing SmartApp? If so, does it improve the current SmartApp?
 - Does it have a good title, description, and configuration preferences? Will the user understand how it works?
 - Does the SmartApp work as expected?
-

Your *Device Handler* could be rejected by SmartThings review team for any of the following reasons:

- The Device Handler adds minor addition or change that may be changed with a core product or UX change in a future update.
 - SmartThings is already developing a first-party integration and will not accept a Device Handler for this device.
 - The Device Handler should actually be a SmartApp instead, because it is actuating or changing a device.
-

- No discovery mechanism is provided. For LAN-Connected devices, a [Service Manager SmartApp](#) should serve to discover and create the device.
 - Multiple community submissions exist and SmartThings is rolling up several improvements together, so this specific one is being rejected.
-

Once your SmartApp or Device Handler has been approved, it will be published for worldwide public distribution in SmartThings mobile app.

Part XVIII

Code Review Guidelines and Best Practices

Before submitting your SmartApp or Device Handler, you should ensure that your code adheres to the guidelines documented here. Any code that does not adhere to these guidelines may be rejected.

This document also serves as a collection of best practices for SmartThings development.

163.1 Code should be readable

Code is executed by machines, but read by humans. Readability can be subjective, but there are some general guidelines that should be followed:

- Use meaningful variable and method names.
- *Don't repeat yourself* (page 633)
- *Methods should serve a single purpose* (page 633)
- *Comment appropriately* (page 634)

163.2 Don't repeat yourself

Follow the **DRY** principle (don't repeat yourself).

Don't copy/paste code blocks - pull common code out into a shared utility method.

163.3 Methods should serve a single purpose

Methods should serve a single purpose, and be concise. If a method definition doesn't fit on a standard computer screen, it's way too big.

Look for opportunities to split out code into utility methods. For example, parsing a large HTTP response inline can bloat a method; instead, split out the parsing into a method that can then be called. This facilitates easier understanding of the code, and promotes better *separation of concerns*.

163.4 Do not submit unused code

Unused or commented-out code should be removed prior to submitting.

163.5 Do not use offensive, profane, or libelous language

This is pretty self-explanatory - language should be clean and professional.

163.6 Comment appropriately

Comments can add clarity and context to code when used appropriately. When over-used, they clutter the code and provide no value.

There are some guidelines that should be followed:

- In general, when the code is doing something out of the ordinary, a comment is appropriate.
- Device Handler custom commands and attributes should have a comment describing the purpose, parameters, and exception conditions (if applicable).
- Non-trivial methods should be documented with comments describing what it does, its return type, exception conditions, and parameters. [JavaDoc style comments](#) can be used, though there is no tooling in place to generate documentation from the source.
- Comments should add value - commenting every line of readable code simply clutters the code and is unnecessary.

Here's an example of using comments appropriately for documenting a method:

```
def capabilityCommands = getDeviceCapabilityCommands(device.capabilities)

/**
 * Builds a map of capability names to their supported commands.
 *
 * @param a list of Capabilities.
 * @return a map of device capability -> supported commands.
 */
def getDeviceCapabilityCommands(deviceCapabilities) {
    def map = [:]
    deviceCapabilities.collect {
        map[it.name] = it.commands.collect { it.name.toString() }
    }
    return map
}
```

Here's an example of an in-line code comment explaining why the code is checking if a percentage value is within a certain hard-coded range:

```
log.trace "stopDimmersHandler evt: ${evt.value}"
def percentComplete = completionPercentage()

// Oftentimes, the first thing we do is turn lights on or off,
// so make sure we don't stop as soon as we start
if (percentComplete > 2 && percentComplete < 98) {
    ...
}
```

An example of inappropriate comments is below. Note how the comments simply repeat what is obvious by reading the code; no value is added.

```
// get all the children
def children = pollChildren()
// iterate over all the children
children.each { child ->
    // log each child
    log.debug "child: $child"
}
```

163.7 Handle all `if()` and `switch()` cases

Make sure any `if()` or `switch()` blocks handle all expected inputs. Forgetting to handle a certain condition can cause unexpected logic errors.

Also, every `switch()` statement should have a `default:` case statement to handle any cases where there is no match.

163.8 Verify assumptions

If a method operates on some input, it should handle all possible input values, including any differences if the method is called from a parent or child SmartApp or Device Handler.

163.9 Use consistent return values

Groovy is a dynamically typed language. That's great for a lot of things, but it's a sharp knife - highly effective, yet also easy to cut yourself accidentally.

A method should return a single type of data, regardless of if the method signature is typed or not. For example, don't do something like this:

```
def getSomeResult(input) {
    if (input == "option1") {
        return true
    }
    if (input == "option2") {
        return false
    }
    return [name: "someAttribute", value: input]
}
```

The example above fails to return a consistent data type. Calling clients of this code have to accommodate both a boolean and map return values. Instead, methods should always return the same data type.

Note: In certain cases, it *may* make sense for a method to return different types. Such cases are the exception, and the different types returned, and under what circumstances, should be documented in the method's comments.

163.10 Be careful indexing into arrays

When parsing data, pay attention to arrays if you use them. Do not index into arrays directly without making sure that the array actually has enough elements.

Consider the following code that splits a string on the `:` character, and returns the value after the `:`:

```
def getSplitString(input) {
    return input.split(":")[1]
}

// -> "123"
getSplitString("abc:123")
```

```
// -> ArrayIndexOutOfBoundsException exception!  
getSplitString("abc:");
```

Because `getSplitString()` does not verify that the result of `split()` split has more than one element, we get an `ArrayIndexOutOfBoundsException` exception when trying to access the second item in the parsed result. In cases like this, make sure your code verifies the array contains the item:

```
def getSplitString(input) {  
    def splitted = input?.split(":")  
    if (splitted?.size() == 2) {  
        return splitted[1]  
    } else {  
        return null  
    }  
}
```

163.11 Use the Elvis operator correctly

Groovy supports the Elvis operator, which allows us write more concise conditional expressions than otherwise possible. However, we need to understand *Groovy truth* (page 637) to use it effectively.

Consider this example that attempts to set the variable `bulbLevel` to 100 if it is not already set:

```
def bulbLevel = settings.level ?: 100
```

But what happens if `settings.level` is 0 in the example above? **Because Groovy considers zero as false, we've set `bulbLevel` to 100!**

The above expression should be rewritten as:

```
def bulbLevel = settings.level == null ?: 100
```

163.12 Handle null values

Important: `NullPointerException`s are one of the most frequently occurring exceptions on the SmartThings platform - take care to avoid them!

This is *very* common in LAN and SSDP interactions, so always double check that code.

A `NullPointerException` will terminate the SmartApp or Device Handler execution, but can be avoided easily with the *safe navigation* (`?`) operator. Any code that may encounter a `null` value should anticipate and handle this.

The examples below show a few common scenarios in which `null` is possible, and how to deal with it using the `?` operator:

```
// if the LAN event does not have headers, or a "content-type" header,  
// don't blow up with a NullPointerException!  
if (lanEvent.headers?. "content-type"?.contains("xml")) { ... }
```

```
// if a location does not have any modes, statement simply returns null  
// but does not throw a NullPointerException  
if (location.modes?.find{ it.name == newMode }) { ... }
```


163.13 Use Groovy truth correctly

Be aware of, and ensure your code is consistent with, what Groovy considers true and false. Groovy truth is documented [here](#).

Here are some gotchas to be aware of:

- Empty strings are considered `false`; non-empty strings are considered `true`.
- Empty maps and lists are considered `false`; non-empty maps and lists are considered `true`.
- Zero is considered `false`; non-zero numbers are considered `true`.

Consider the following example that verifies that a number is between 0 and 100:

```
def verifyLevel(level) {
    if (!level) {
        return false
    } else {
        return (level >= 0 && level <= 100)
    }
}
```

If we call `verifyLevel(0)`, the result is `false`, because 0 is treated as false by Groovy. Instead, it should be written as:

```
def verifyLevel(level) {
    return (level instanceof Number && level >= 0 && level <= 100)
}
```

This can be a common source of errors; make sure you understand and use Groovy truth appropriately.

Using State

164.1 `state` is not an unbounded database

`state` (SmartApps and Device Handlers) and `atomicState` (SmartApps only) are provided to persist small amounts of data across executions. Do not think of state as a virtually unlimited database for your app.

The amount of data that can be stored in state is *limited* (page 323). Avoid code that adds items to `state` regularly (perhaps in response to Events or schedules), but does not remove items.

164.2 Understand how `state` works

Remember that when using `state`, the *results are not persisted until the app is done executing* (page 317). This can have unintended consequences, such as state values being overridden by another concurrently executing instance of the SmartApp.

164.3 Understand when to use `atomicState` vs. `state`

Understand the *difference* (page 318) between `atomicState` and `state`, make sure you use the correct one for your needs, and avoid using both in the same SmartApp.

164.4 Take care when storing collections in `atomicState`

Modifying collections in Atomic State does not work as it does with State. *Read the documentation* (page 323) to understand how to best work with collections stored in Atomic State.

Web Services

165.1 Document external HTTP requests

HTTP requests (page 365) to outside services should be documented, explaining the need to make external requests, what data is sent, and how it will be used. Please also include a comment with a link to the third party's privacy policy, if applicable.

165.2 Document any exposed endpoints

If your SmartApp or Device Handler *exposes any endpoints* (page 435), add comments that document what the API will be used for, what data may be accessed by those APIs, and where possible, include a link to the privacy policies of any remote services that may access those APIs.

Scheduling

166.1 Avoid recurring short schedules

Scheduled and other periodic functions should not execute more often than every five minutes, unless there is a good reason for it, and the reviewers agree.

If your code executes more frequently than every five minutes, add a comment to your code explaining why this is necessary.

166.2 Avoid chained `runIn()` calls

Do not chain `runIn()` calls (page 353).

If for some reason it is necessary, add a comment describing why it is necessary.

Security considerations

167.1 Subscriptions should be clear

It is possible to subscribe to Events using a string variable, so what the SmartApp is subscribing to might be somewhat opaque.

For example:

```
def myContactSubscription = "contact.open"
...
subscribe(contact1, myContactSubscription, myContactHandler)
```

The best practice is to subscribe explicitly to the attribute:

```
subscribe(contact1, "contact.open", myContactHandler)
```

However, if the SmartApp must subscribe to a variable (from state, for instance), the reviewer should be able to trace how the variable is set and what the expected attribute will be.

167.2 Subscriptions should be specific

Do not create overly-broad subscriptions.

A SmartApp that is subscribed to every location Event will execute excessively, and is rarely necessary. Instead, create subscriptions specific to the Event you are interested in.

If you're creating a service manager for a LAN-connected device, be sure to *subscribe to the device search target* (page 563).

167.3 Do not use dynamic method execution

In groovy you can execute functions based on a string, like so:

```
object."${mystring}"()
```

Which can be very handy, but when `${mystring}` comes from a HTTP request, outside the SmartThings platform, or from another SmartApp or Device Handler, we need to validate the input.

The preferred method of validation is to use a `switch()` statement on the input before doing anything with it:

```
switch(mystring) {
  case "cmd1":
    object.cmd1()
    break
  case "cmd2":
    object.cmd2()
    break
  case "cmd3":
    object.cmd3()
    break
  default:
    return "ERROR"
}
```

167.4 Do not hard-code SMS messages

Notifications should never be sent to a hard-coded number. They should always use a number provided by the user using the *contact input* (page 385) (even though Contact Book is not enabled, the contact input type is available and contains a fall-back mechanism for non-Contact Book users. Using this future-proofs your SmartApp).

Performance

168.1 Do not use busy loops

There is no good reason for the code to run busy loops. Don't do things like this:

```
def mywait(ms) {
  def start = now()
  while (now() < start + ms) {
    // do nothing, just wait
  }
}
```

The goal of the above code is to delay execution for a number of milliseconds. This wastes resources and increases the likelihood that the 20 second execution limit will be exceeded.

Instead of trying to force a delay in execution, you should *schedule* (page 345) a future execution of your app.

168.2 Do not use `synchronized()`

Using `synchronized` incurs a performance overhead, and is highly unlikely to have any effect. It should not be used.

When a SmartApp or Device Handler executes, it is executing on one of n available servers assigned for that Location, where n is variable depending on Location, current load, and other factors. Concurrent executions of the SmartApp or Device Handler are not guaranteed, or even likely, to be executing on the same server. Because of this, trying to force synchronous behavior by using `synchronized` would only work in the rare occurrence that a concurrent execution happens on the same server, yet it always incurs overhead.

LAN-specific

169.1 Use the device-specific search

Service managers for LAN-connected devices should *subscribe to the device search target* (page 563) for device discovery.

169.2 Handle IP change

Service managers for LAN-connected devices should *handle any IP change* (page 565). This can happen when the router power cycles and loses its DHCP mappings.

Parent-child relationships

170.1 Use separate files

When using a parent-child relationship, be it a parent SmartApp with child devices, or a parent SmartApp with child SmartApps, the parent and child should exist in separate files.

Putting the parent and child code in the same file leads to file size bloat, makes the code harder to understand, is error-prone, and difficult to debug.

Part XIX

Capabilities Reference

Important: The Capabilities in this document are supported in the SmartThings Classic mobile app. Visit the [SmartThings Developer Portal](#) for the Capabilities supported in the new SmartThings app.

Introduction

Capabilities are core to the SmartThings architecture. They allow us to abstract specific devices into their underlying capabilities. An application interacts with devices based on their capabilities, so once we understand the capabilities that are needed by a SmartApp, and the capabilities that are provided by a device, we can understand which devices (based on the Device's declared capabilities) are eligible for use within a specific SmartApp. Capabilities themselves are decomposed into both Commands and Attributes. Commands represent ways in which you can control or actuate the device, whereas Attributes represent state information or properties of the device. Capabilities are created and maintained by the SmartThings development team. This page serves as a reference for the supported capabilities.

Data Types

Before we present the Capabilities, it's worth covering the various data types associated with Attributes and Commands. Note that these data types are guidelines as to how actual values can be represented. In most cases, the SmartThings platform contains the implementation logic and defines the actual objects for these data types. Below is a table outlining the the possible data types and what they mean.

Data Type	Example	Description
STRING	"This is a String"	Represents character strings
NUMBER	5, 10.67	The Number data type is a guideline indicating that a number should be expected, and not a specific type. Device Handlers contain the implementation of what kind of number object is actually returned.
VECTOR3	(x, y, z)	This Data Type is a representation of x,y,z coordinates in space. Device Handlers contain the implementation of the actual data structure, but it is usually as a Map: [x: 0, y: 0, z: 0].
ENUM	"one", "two", "three"	The Enum Data Type is a static set of predefined String values that an Attribute can have, or that a Command can accept as an argument.
DYNAMIC_ENUM	"Any", "value"	Much like the Enum Data Type, Dynamic Enum is a set of String values. However, the set is not static or predefined.
COLOR_MAP	hue: 50, saturation: 75]	The Color Map is a Map specifically for the use of color control. As such, the Map should contain a Hue and a Saturation value.
JSON_OBJECT		A standard JSON object. Device Handlers contain the implementation and thus should be consulted when looking for the JSON object structure.
DATE		A Date, usually represented as a java.util.Date object.
BOOLEAN	true, false	A boolean data type with a value of true or false.

Acceleration Sensor

The Acceleration Sensor capability allows for acceleration detection. Some use cases for SmartApps using this capability would be detecting if a washing machine is vibrating, or if a case has moved (particularly useful for knowing if a weapon case has been moved).

173.1 Definition

```
# reviewed 2018-01-03
name: Acceleration Sensor
status: live
attributes:
  acceleration:
    schema:
      type: object
      properties:
        value:
          $ref: ActivityState
      required: ["value"]
    type: ENUM
    values:
      - active
      - inactive
commands: {}
public: true
id: accelerationSensor
ocfResourceType: x.com.st.acceleration
version: 1
```

Actuator

The Actuator capability is a “tagging” capability. It defines no attributes or commands. In SmartThings terms, it represents that a Device has commands.

174.1 Definition

```
# reviewed 2018-02-01
name: Actuator
status: deprecated
attributes: {
}
commands: {
}
public: true
id: actuator
version: 1
```

Air Conditioner Mode

Allows for the control of the air conditioner.

175.1 Definition

```
# reviewed 2018-01-03
name: Air Conditioner Mode
status: proposed
attributes:
  airConditionerMode:
    schema:
      type: object
      properties:
        value:
          $ref: HvacMode
      required: ["value"]
    type: ENUM
    values:
      - auto
      - cool
      - dry
      - coolClean
      - dryClean
      - fanOnly
      - heat
      - heatClean
      - notSupported
    setter: setAirConditionerMode
commands:
  setAirConditionerMode:
    arguments:
      - name: mode
        required: true
    schema:
      $ref: HvacMode
    type: ENUM
    values:
      - auto
      - cool
      - dry
      - coolClean
```

```
- dryClean
- fanOnly
- heat
- heatClean
- notSupported
public: true
id: airConditionerMode
ocfResourceType: x.com.st.mode.airconditioner
version: 1
```

Air Quality Sensor

Gets the air quality number.

176.1 Definition

```
# reviewed 2018-01-11
name: Air Quality Sensor
status: proposed
attributes:
  airQuality:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
          required: ["value"]
      type: NUMBER
commands: (
)
public: true
id: airQualitySensor
ocfResourceType: x.com.st.airqualitylevel
version: 1
```

Alarm

The Alarm capability allows for interacting with devices that serve as alarms

177.1 Definition

```
# reviewed 2018-01-03
name: Alarm
status: live
attributes:
  alarm:
    schema:
      type: object
      properties:
        value:
          $ref: AlertState
      required: ["value"]
    type: ENUM
    values:
      - both
      - 'off'
      - siren
      - strobe
    enumCommands:
      - command: both
        value: both
      - command: 'off'
        value: 'off'
      - command: siren
        value: siren
      - command: strobe
        value: strobe
  commands:
    both:
      arguments: {}
    'off':
      arguments: {}
    siren:
      arguments: {}
```

```
  strobe:
    arguments: |
public: true
id: alarm
ocfResourceType: x.com.st.alarm
version: 1
```

Audio Mute

Allows for the control of audio mute.

178.1 Definition

```
# reviewed 2018-02-01
name: Audio Mute
status: live
attributes:
  mute:
    schema:
      type: object
      properties:
        value:
          $ref: MuteState
      required:
        - value
    type: ENUM
    values:
      - muted
      - unmuted
    setter: setMute
    enumCommands:
      - command: mute
        value: muted
      - command: unmute
        value: unmuted
commands:
  setMute:
    arguments:
      - name: state
        required: true
    schema:
      $ref: MuteState
    type: ENUM
    values:
      - muted
      - unmuted
  mute:
    arguments: [
```

```
unmute:
  arguments: []
public: true
id: audioMute
version: 1
```

Audio Notification

Play a track or a message as an audio notification

179.1 Definition

```
# reviewed 2018-01-03
name: Audio Notification
status: proposed
attributes: {
}
commands:
  playTrack:
    arguments:
      - name: uri
        required: true
        schema:
          $ref: URI
        type: STRING
      - name: level
        schema:
          $ref: IntegerPercent
        type: NUMBER
        required: false
  playTrackAndResume:
    arguments:
      - name: uri
        required: true
        schema:
          $ref: URI
        type: STRING
      - name: level
        schema:
          type: integer
          minimum: 0
          maximum: 100
        type: NUMBER
        required: false
  playTrackAndRestore:
    arguments:
      - name: uri
        required: true
```

```
    schema:
      $ref: URI
      type: STRING
  - name: level
    schema:
      type: integer
      minimum: 0
      maximum: 100
      type: NUMBER
      required: false
public: true
id: audioNotification
ocfResourceType: x.com.st.audionotification
version: 1
```

Audio Track Data

Gets the value of the audio track data.

180.1 Definition

```
# reviewed 2018-02-01
name: Audio Track Data
status: proposed
attributes:
  audioTrackData:
    schema:
      type: object
      properties:
        value:
          $ref: AudioTrackAddress
      required:
        - value
    type: JSON_OBJECT
commands: (
)
public: true
id: audioTrackData
ocfResourceType: x.com.st.audiotrackdata
version: 1
```

Audio Volume

Allows for the control of audio volume.

181.1 Definition

```
# reviewed 2018-01-03
name: Audio Volume
status: proposed
attributes:
  volume:
    schema:
      $ref: IntegerPercent
    type: NUMBER
    setter: setVolume
    actedOnBy:
      - volumeUp
      - volumeDown
commands:
  setVolume:
    arguments:
      - name: volume
        required: true
        schema:
          type: integer
          minimum: 0
          maximum: 100
        type: NUMBER
  volumeUp:
    arguments: []
  volumeDown:
    arguments: []
public: true
id: audioVolume
ocfResourceType: x.com.st.audiovolume
version: 1
```

Battery

Defines that the device has a battery

182.1 Definition

```
# reviewed 2018-01-03
name: Battery
status: live
attributes:
  battery:
    schema:
      $ref: IntegerPercent
    type: NUMBER
    unit: '%'
commands: {}
public: true
id: battery
ocfResourceType: oic.r.energy.battery
version: 1
```

Beacon

Detect whether or not the beacon is present

183.1 Definition

```
# reviewed 2018-02-01
name: Beacon
status: deprecated
attributes:
  presence:
    schema:
      type: object
      properties:
        value:
          $ref: PresenceState
      required:
        - value
    type: ENUM
    values:
      - not present
      - present
commands: (
)
public: true
id: beacon
version: 1
```

Bridge

The Bridge capability is a “tagging” capability. It defines no attributes or commands. In SmartThings terms, it represents that a Device is a bridge to other devices.

184.1 Definition

```
# reviewed 2018-02-01
name: Bridge
status: deprecated
attributes: {
}
commands: {
}
public: true
id: bridge
version: 1
```

Bulb

Allows for the control of a bulb device

185.1 Definition

```
# reviewed 2018-01-11
name: Bulb
status: dead
attributes:
  switch:
    schema:
      type: object
      properties:
        value:
          $ref: SwitchState
      required: ["value"]
    type: ENUM
    values:
      - 'off'
      - 'on'
    enumCommands:
      - command: 'off'
        value: 'off'
      - command: 'on'
        value: 'on'
  commands:
    'off':
      arguments: {}
    'on':
      arguments: {}
public: true
id: bulb
version: 1
```

Button

A device with one or more buttons

186.1 Definition

```
# reviewed 2018-02-22
name: Button
status: deprecated
attributes:
  button:
    schema:
      type: object
      properties:
        value:
          $ref: ButtonState
      required:
        - value
    type: ENUM
    values:
      - held
      - pushed
  numberOfButtons:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
      required:
        - value
    type: NUMBER
commands: (
)
public: true
id: button
ocfResourceType: x.com.st.button
version: 1
```

Carbon Dioxide Measurement

Measure carbon dioxide levels

187.1 Definition

```
# reviewed 2018-02-20
name: Carbon Dioxide Measurement
status: live
attributes:
  carbonDioxide:
    schema:
      type: object
      properties:
        value:
          type: integer
          minimum: 0
          maximum: 1000000
        unit:
          type: string
          enum:
            - ppm
          default:
            - ppm
      required:
        - value
    type: NUMBER
commands: {}
public: true
id: carbonDioxideMeasurement
version: 1
```

Carbon Monoxide Detector

Measure carbon monoxide levels

188.1 Definition

```
# reviewed 2018-01-09
name: Carbon Monoxide Detector
status: live
attributes:
  carbonMonoxide:
    schema:
      type: object
      properties:
        value:
          $ref: CarbonMonoxideState
      required: ["value"]
    type: ENUM
    values:
      - clear
      - detected
      - tested
commands: {}
public: true
id: carbonMonoxideDetector
ocfResourceType: x.com.st.carbonmonoxidedetector
version: 1
```

Color Control

Allows for control of a color changing device by setting its hue, saturation, and color values

189.1 Definition

```
# reviewed 2018-01-16
name: Color Control
status: live
attributes:
  color:
    schema:
      type: object
      properties:
        value:
          $ref: String
    type: STRING
    setter: setColor
    actedOnBy:
      - setHue
      - setSaturation
  hue:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
    type: NUMBER
    setter: setHue
  saturation:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
    type: NUMBER
    setter: setSaturation
commands:
  setColor:
    arguments:
      - name: color
        required: true
```

```
type: COLOR_MAP
schema:
  type: object
  properties:
    value:
      $ref: color-map
setHue:
  arguments:
    - name: hue
      required: true
      schema:
        $ref: PositiveInteger
      type: NUMBER
setSaturation:
  arguments:
    - name: saturation
      required: true
      schema:
        $ref: PositiveInteger
      type: NUMBER
public: true
id: colorControl
ocfResourceType: oic.r.colour.chroma
version: 1
```

Color Temperature

Set the color temperature attribute of a color changing device

190.1 Definition

```
# reviewed 2018-01-16
name: Color Temperature
status: live
attributes:
  colorTemperature:
    schema:
      type: object
      properties:
        value:
          type: integer
          minimum: 1
          maximum: 30000
        unit:
          type: string
          enum:
            - K
          default: K
      required:
        - value
    type: NUMBER
    setter: setColorTemperature
commands:
  setColorTemperature:
    arguments:
      - name: temperature
        required: true
        schema:
          type: integer
          minimum: 1
          maximum: 30000
        type: NUMBER
public: true
id: colorTemperature
ocfResourceType: x.com.st.color.temperature
version: 1
```

Color

Allows for control of a color changing device by setting its hue and saturation.

191.1 Definition

```
# reviewed 2018-01-16
id: color
status: proposed
public: true
name: Color
attributes:
  colorValue:
    schema:
      type: object
      properties:
        value:
          type: object
          properties:
            hue:
              type: number
              minimum: 0.0
              maximum: 360.0
            saturation:
              type: number
              minimum: 0.0
              maximum: 100.0
      required:
        - value
    setter: setColorValue
    type: JSON_OBJECT
commands:
  setColorValue:
    arguments:
      - name: colorValue
        required: true
    schema:
      type: object
      properties:
        hue:
          type: number
          minimum: 0.0
```

```
    maximum: 360.0
  saturation:
    type: number
    minimum: 0.0
    maximum: 100.0
  required:
    - hue
    - saturation
  type: JSON_OBJECT
- name: switchLevel
  schema:
    type: integer
    minimum: 0
    maximum: 100
  type: NUMBER
  required: false
version: 1
```

Color Mode

Describes if a device is in color or color temperature mode if it supports both since state is mutually exclusive.

192.1 Definition

```
# reviewed 2018-01-16
id: colorMode
status: proposed
public: true
name: Color Mode
attributes:
  colorMode:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - color
            - colorTemperature
            - other
      type: ENUM
    values:
      - color
      - colorTemperature
      - other
commands: {}
version: 1
```

Configuration

Allow configuration of devices that support it

193.1 Definition

```
# reviewed 2018-02-20
name: Configuration
status: live
attributes: {
}
commands:
  configure:
    arguments: [
]
public: true
id: configuration
version: 1
```

Consumable

For devices with replaceable components

194.1 Definition

```
# reviewed 2018-02-20
name: Consumable
status: proposed
attributes:
  consumableStatus:
    schema:
      type: object
      properties:
        value:
          $ref: ConsumableState
      required:
        - value
    type: ENUM
    values:
      - good
      - maintenance_required
      - missing
      - order
      - replace
    setter: setConsumableStatus
commands:
  setConsumableStatus:
    arguments:
      - name: status
        required: true
    schema:
      $ref: ConsumableState
    type: ENUM
    values:
      - good
      - maintenance_required
      - missing
      - order
      - replace
public: true
id: consumable
version: 1
```

Contact Sensor

Allows reading the value of a contact sensor device

195.1 Definition

```
# reviewed 2018-01-09
name: Contact Sensor
status: live
attributes:
  contact:
    schema:
      type: object
      properties:
        value:
          $ref: ContactState
      required: ["value"]
    type: ENUM
    values:
      - closed
      - open
commands: (
)
public: true
id: contactSensor
ocfResourceType: oic.r.sensor.contact
version: 1
```

Demand Response Load Control

Allows requests to be made to appliances to temporarily reduce their energy usage to reduce demand on the power grid

196.1 Definition

```
name: Demand Response Load Control
status: proposed
attributes:
  drlcStatus:
    schema:
      type: object
      properties:
        value:
          $ref: DemandResponseLoadControlStatus
      required: ["value"]
    type: JSON_OBJECT
commands:
  requestDrlcAction:
    arguments:
      - name: drlcType
        required: true
        schema:
          $ref: DrlcType
        type: NUMBER
      - name: drlcLevel
        required: true
        schema:
          $ref: DrlcLevel
        type: NUMBER
      - name: start
        required: true
        schema:
          $ref: Iso8601Date
        type: STRING
      - name: duration
        required: true
        schema:
          $ref: PositiveInteger
        type: NUMBER
      - name: reportingPeriod
```

```
    schema:
      $ref: PositiveInteger
      type: NUMBER
      required: false
  overrideDrlcAction:
    arguments:
      - name: value
        required: true
        schema:
          type: boolean
        type: BOOLEAN

public: true
id: demandResponseLoadControl
ocfResourceType: oic.r.energy.drlc #https://oneiota.org/revisions/1761
version: 1
```

Dishwasher Mode

Allows for the control of the dishwasher.

197.1 Definition

```
name: Dishwasher Mode
status: proposed
attributes:
  dishwasherMode:
    schema:
      type: object
      properties:
        value:
          $ref: DishwasherMode
      required: ["value"]
    type: ENUM
    values:
      - auto
      - quick
      - rinse
      - dry
    setter: setDishwasherMode
commands:
  setDishwasherMode:
    arguments:
      - name: mode
        required: true
    schema:
      $ref: DishwasherMode
    type: ENUM
    values:
      - auto
      - quick
      - rinse
      - dry
public: true
id: dishwasherMode
ocfResourceType: x.com.st.mode.dishwasher
version: 1
```

Dishwasher Operating State

Allows for the control of the dishwasher operational state.

198.1 Definition

```
name: Dishwasher Operating State
status: proposed
attributes:
  machineState:
    schema:
      type: object
      properties:
        value:
          $ref: MachineState
      constraints:
        type: object
        properties:
          values:
            type: array
            items:
              $ref: MachineState
        required: ["value"]
    type: ENUM
    values:
      - pause
      - run
      - stop
    setter: setMachineState
supportedMachineStates:
  schema:
    type: object
    properties:
      value:
        type: array
        items:
          $ref: MachineState
    requires:
      - value
  type: JSON_OBJECT
dishwasherJobState:
  schema:
```

```
type: object
properties:
  value:
    $ref: DishwasherJobState
  constraints:
    type: object
    properties:
      values:
        type: array
        items:
          $ref: DishwasherJobState
    required: ["value"]
type: ENUM
values:
  - airwash
  - cooling
  - drying
  - finish
  - preDrain
  - prewash
  - rinse
  - spin
  - unknown
  - wash
  - wrinklePrevent
completionTime:
  schema:
    type: object
    properties:
      value:
        $ref: Iso8601Date
    required:
      - value
  type: DATE
commands:
  setMachineState:
    arguments:
      - name: state
        required: true
    schema:
      $ref: MachineState
    type: ENUM
    values:
      - pause
      - run
      - stop
public: true
id: dishwasherOperatingState
ocfResourceType: x.com.st.operationalstate.dishwasher
version: 1
```

Door Control

Allow for the control of a door

199.1 Definition

```
# reviewed 2018-02-20
name: Door Control
status: live
attributes:
  door:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - closed
            - closing
            - open
            - opening
            - unknown
      constraints:
        type: object
        properties:
          values:
            type: array
            items:
              type: string
              enum:
                - closed
                - closing
                - open
                - opening
                - unknown
        required:
          - value
    type: ENUM
    values:
      - closed
      - closing
      - open
```

```
- opening
- unknown
enumCommands:
- command: close
  value: closed
- command: open
  value: open
commands:
  close:
    arguments: [
    ]
  open:
    arguments: [
    ]
public: true
id: doorControl
ocfResourceType: x.com.st.doorcontrol
version: 1
```

Dryer Mode

Allows for the control of the dryer.

200.1 Definition

```
name: Dryer Mode
status: proposed
attributes:
  dryerMode:
    schema:
      type: object
      properties:
        value:
          $ref: DryerMode
      constraints:
        type: object
        properties:
          values:
            type: array
            items:
              $ref: DryerMode
        required: ["value"]
    type: ENUM
    values:
      - regular
      - lowHeat
      - highHeat
    setter: setDryerMode
commands:
  setDryerMode:
    arguments:
      - name: mode
        required: true
    schema:
      $ref: DryerMode
    type: ENUM
    values:
      - regular
      - lowHeat
      - highHeat
public: true
```

```
id: dryerMode  
ocfResourceType: x.com.st.mode.dryer  
version: 1
```

Dryer Operating State

Allows for the control of the dryer operational state.

201.1 Definition

```
name: Dryer Operating State
status: proposed
attributes:
  machineState:
    schema:
      type: object
      properties:
        value:
          $ref: MachineState
        constraints:
          type: object
          properties:
            values:
              type: array
              items:
                $ref: MachineState
      required: ["value"]
    type: ENUM
    values:
      - pause
      - run
      - stop
    setter: setMachineState
  supportedMachineStates:
    schema:
      type: object
      properties:
        value:
          type: array
          items:
            $ref: MachineState
      required: ["value"]
    type: JSON_OBJECT
  dryerJobState:
    schema:
      type: object
```

```
properties:
  value:
    $ref: DryerJobState
  constraints:
    type: object
  properties:
    values:
      type: array
      items:
        $ref: DryerJobState
  required: ["value"]
type: ENUM
values:
- cooling
- delayWash
- drying
- finished
- none
- weightSensing
- wrinklePrevent
completionTime:
  schema:
    type: object
  properties:
    value:
      $ref: Iso8601Date
  required:
  - value
  type: DATE
commands:
  setMachineState:
    arguments:
      - name: state
        required: true
    schema:
      $ref: MachineState
    type: ENUM
    values:
      - pause
      - run
      - stop
public: true
id: dryerOperatingState
ocfResourceType: x.com.st.operationalstate.dryer
version: 1
```

Dust Sensor

Gets the reading of the dust sensor.

202.1 Definition

```
name: Dust Sensor
status: proposed
attributes:
  fineDustLevel:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
      required: ["value"]
    type: NUMBER
  dustLevel:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
      required: ["value"]
    type: NUMBER
commands: (
)
public: true
id: dustSensor
ocfResourceType: x.com.st.dustlevel
version: 1
```

Energy Meter

Read the energy consumption of an energy metering device

203.1 Definition

```
# reviewed 2018-02-20
name: Energy Meter
status: live
attributes:
  energy:
    schema:
      type: object
      properties:
        value:
          type: number
        unit:
          type: string
          enum:
            - kWh
          default: kWh
      required:
        - value
    type: NUMBER
commands: (
)
public: true
id: energyMeter
ocfResourceType: x.com.st.energymeter
version: 1
```

Estimated Time Of Arrival

Allow access to estimated time of arrival values for devices that support it, for example automobiles

204.1 Definition

```
# reviewed 2018-02-20
name: Estimated Time Of Arrival
status: proposed
attributes:
  eta:
    schema:
      type: object
      properties:
        value:
          $ref: Iso8601Date
      required:
        - value
    type: DATE
commands: (
)
public: true
id: estimatedTimeOfArrival
version: 1
```

Execute

Allows for raw messages to be passed to a device.

205.1 Definition

```
name: Execute
status: proposed
attributes:
  data:
    schema:
      type: object
      properties:
        value:
          $ref: JsonObject
      required:
        - value
    type: JSON_OBJECT
    actedOnBy:
      - 'execute'
commands:
  'execute':
    arguments:
      - name: command
        required: true
        type: STRING
        schema:
          $ref: String
      - name: args
        schema:
          $ref: JsonObject
        type: JSON_OBJECT
        required: false
public: true
id: execute
version: 1
```

Fan Speed

Allows for the control of the fan speed.

206.1 Definition

```
name: Fan Speed
status: proposed
attributes:
  fanSpeed:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
          required: ["value"]
      type: NUMBER
      setter: setFanSpeed
commands:
  setFanSpeed:
    arguments:
      - name: speed
        required: true
        schema:
          $ref: PositiveInteger
        type: NUMBER
public: true
id: fanSpeed
ocfResourceType: x.com.st.fanspeed
version: 1
```

Filter Status

Gets the status of the filter.

207.1 Definition

```
name: Filter Status
status: proposed
attributes:
  filterStatus:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - normal
            - replace
          required: ["value"]
      type: ENUM
      values:
        - normal
        - replace
    commands: {}
public: true
id: filterStatus
ocfResourceType: x.com.st.filter
version: 1
```

Garage Door Control

Allow for the control of a garage door. Deprecated in favor of Door Control.

208.1 Definition

```
# reviewed 2018-02-20
name: Garage Door Control
status: deprecated
attributes:
  door:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - closed
            - closing
            - open
            - opening
            - unknown
        constraints:
          type: object
          properties:
            values:
              type: array
              items:
                type: string
                enum:
                  - closed
                  - closing
                  - open
                  - opening
                  - unknown
      required:
        - value
type: ENUM
values:
  - closed
  - closing
  - open
```

```
- opening
- unknown
enumCommands:
  - command: close
    value: closed
  - command: open
    value: open
commands:
  close:
    arguments: [
    ]
  open:
    arguments: [
    ]
public: true
id: garageDoorControl
ocfResourceType: x.com.st.garagedoorcontrol
version: 1
```

Geolocation

Gets the value of the geo location.

209.1 Definition

```
id: geolocation
name: Geolocation
status: proposed
public: true
attributes:
  latitude:
    schema:
      type: object
      properties:
        value:
          type: integer
          maximum: 90
          minimum: -90
      type: NUMBER
  longitude:
    schema:
      type: object
      properties:
        value:
          type: integer
          maximum: 180
          minimum: -180
      type: NUMBER
  method:
    schema:
      type: object
      properties:
        value:
          $ref: String
      type: STRING
  accuracy:
    schema:
      type: object
      properties:
        value:
          type: number
```

```
        minimum: 0
        # maximum: ??
    type: NUMBER
altitudeAccuracy:
  schema:
    type: object
    properties:
      value:
        type: number
        minimum: 0
        # maximum: ??
    type: NUMBER
heading:
  schema:
    type: object
    properties:
      value:
        type: number
        minimum: 0
        maximum: 360
    type: NUMBER
speed:
  schema:
    type: object
    properties:
      value:
        type: number
        minimum: 0
        # maximum: ??
    type: NUMBER
lastUpdateTime:
  schema:
    type: object
    properties:
      value:
        $ref: PositiveInteger
    type: NUMBER
commands: {
}
version: 1
```

Holdable Button

A device with one or more holdable buttons. Deprecated in favor of Button.

210.1 Definition

```
# reviewed 2018-2-20
name: Holdable Button
status: deprecated
attributes:
  button:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - held
            - pushed
      required:
        - value
    type: ENUM
    values:
      - held
      - pushed
  numberOfButtons:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
      required:
        - value
    type: NUMBER
commands: (
)
public: true
id: holdableButton
version: 1
```

Illuminance Measurement

Gives the illuminance reading from devices that support it

211.1 Definition

```
# reviewed 2018-01-09
name: Illuminance Measurement
status: live
attributes:
  illuminance:
    schema:
      type: object
      properties:
        value:
          type: number
          minimum: 0
          maximum: 100000
        unit:
          type: string
          enum:
            - lux
          default: lux
      required: ["value"]
    type: NUMBER
    unit: lux
  commands: (
)
public: true
id: illuminanceMeasurement
ocfResourceType: oic.r.sensor.illuminance
version: 1
```

Image Capture

Allows for the capture of an image on devices that support it

212.1 Definition

```
# reviewed 2018-2-20
name: Image Capture
status: proposed
attributes:
  image:
    schema:
      type: object
      properties:
        value:
          $ref: URL
      required:
        - value
      type: STRING
      setter: take
commands:
  take:
    arguments: {
      |
public: true
id: imageCapture
ocfResourceType: x.com.st.imagecapture
version: 1
```

Indicator

The indicator capability gives you the ability to set the indicator LED light on a Z-Wave switch. As such, the most common use case for the indicator capability is in a Device Handler.

213.1 Definition

```
# reviewed 2018-2-20
name: Indicator
status: deprecated
attributes:
  indicatorStatus:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - never
            - when off
            - when on
      required:
        - value
    type: ENUM
    values:
      - never
      - when off
      - when on
    enumCommands:
      - command: indicatorNever
        value: never
      - command: indicatorWhenOff
        value: when off
      - command: indicatorWhenOn
        value: when on
  commands:
    indicatorNever:
      arguments: [ ]
    indicatorWhenOff:
      arguments: [ ]
```

```
indicatorWhenOn:  
  arguments: []  
  |  
public: true  
id: indicator  
version: 1
```

Infrared Level

Allows for the control of the infrared level attribute of a device

214.1 Definition

```
# reviewed 2018-2-20
name: Infrared Level
status: live
attributes:
  infraredLevel:
    schema:
      $ref: Percent
    type: NUMBER
    setter: setInfraredLevel
commands:
  setInfraredLevel:
    arguments:
      - name: level
        required: true
        schema:
          type: number
          minimum: 0
          maximum: 100
        type: NUMBER
public: true
id: infraredLevel
version: 1
```

Light

Allows for the control of a light device

215.1 Definition

```
# reviewed 2018-01-11
name: Light
status: deprecated
attributes:
  switch:
    schema:
      type: object
      properties:
        value:
          $ref: SwitchState
      required: ["value"]
    type: ENUM
    values:
      - 'off'
      - 'on'
    enumCommands:
      - command: 'on'
        value: 'on'
      - command: 'off'
        value: 'off'
  commands:
    'off':
      arguments: {}
    'on':
      arguments: {}
public: true
id: light
version: 1
```

Lock Only

Allow for the lock control of a lock device

216.1 Definition

```
# reviewed 2018-02-22
name: Lock Only
status: deprecated
attributes:
  lock:
    schema:
      type: object
      properties:
        value:
          $ref: LockState
      required:
        - value
    type: ENUM
    values:
      - locked
      - unknown
      - unlocked
      - unlocked with timeout
    enumCommands:
      - command: lock
        value: locked
commands:
  lock:
    arguments: [ ]
public: true
id: lockOnly
version: 1
```


Allow for the control of a lock device

217.1 Definition

```
# reviewed 2018-02-22
name: Lock
status: proposed
attributes:
  lock:
    schema:
      type: object
      properties:
        value:
          $ref: LockState
        data:
          type: object
          properties:
            method:
              type: string
            enum:
              - manual
              - keypad
              - auto
              - command
            codeId:
              type: string
            timeout:
              $ref: Iso8601Date
          required:
            - value
      type: ENUM
      values:
        - locked
        - unknown
        - unlocked
        - unlocked with timeout
      enumCommands:
        - command: lock
          value: locked
        - command: unlock
```

```
        value: unlocked
commands:
  lock:
    arguments: []
  unlock:
    arguments: []
public: true
id: lock
ocfResourceType: oic.r.lock.status
version: 1
```

Media Controller

Allows for the control of a media controller device

218.1 Definition

```
# reviewed 2018-02-22
name: Media Controller
status: proposed
attributes:
  activities:
    schema:
      type: object
      properties:
        value:
          $ref: JsonObject
      required:
        - value
    type: JSON_OBJECT
    actedOnBy:
      - startActivity
  currentActivity:
    schema:
      type: object
      properties:
        value:
          $ref: String
      required:
        - value
    type: STRING
    actedOnBy:
      - startActivity
commands:
  startActivity:
    arguments:
      - type: STRING
      required: true
    schema:
      $ref: String
    name: activityId
public: true
id: mediaController
version: 1
```

Media Input Source

Allows for the control of the media input source.

219.1 Definition

```
name: Media Input Source
status: proposed
attributes:
  inputSource:
    schema:
      type: object
      properties:
        value:
          $ref: MediaSource
      required: ["value"]
    type: ENUM
    values:
      - AM
      - CD
      - FM
      - HDMI
      - HDMI2
      - USB
      - YouTube
      - aux
      - bluetooth
      - digital
      - melon
      - wifi
    setter: setInputSource
supportedInputSources:
  schema:
    type: object
    properties:
      value:
        type: array
        items:
          $ref: MediaSource
    required: ["value"]
type: JSON_OBJECT
```

```
commands:
  setInputSource:
    arguments:
      - name: mode
        required: true
        schema:
          $ref: MediaSource
        type: ENUM
        values:
          - AM
          - CD
          - FM
          - HDMI
          - HDMI2
          - USB
          - YouTube
          - aux
          - bluetooth
          - digital
          - melon
          - wifi
    public: true
  id: mediaInputSource
  ocfResourceType: x.com.st.mediainputsource
  version: 1
```

Media Playback Repeat

Allows for the control of the media playback repeat.

220.1 Definition

```
name: Media Playback Repeat
status: proposed
attributes:
  playbackRepeatMode:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - all
            - 'off'
            - one
          required: ["value"]
    type: ENUM
    values:
      - all
      - 'off'
      - one
    setter: setPlaybackRepeatMode
commands:
  setPlaybackRepeatMode:
    arguments:
      - name: mode
        required: true
        schema:
          type: string
          enum:
            - all
            - 'off'
            - one
        type: ENUM
    values:
      - all
      - 'off'
      - one
```

```
public: true
id: mediaPlaybackRepeat
ocfResourceType: x.com.st.mediarepeat
version: 1
```

Media Playback Shuffle

Allows for the control of media playback shuffle.

221.1 Definition

```
name: Media Playback Shuffle
status: proposed
attributes:
  playbackShuffle:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - disabled
            - enabled
          required: ["value"]
      type: ENUM
      values:
        - disabled
        - enabled
    setter: setPlaybackShuffle
commands:
  setPlaybackShuffle:
    arguments:
      - name: shuffle
        required: true
    schema:
      type: string
      enum:
        - disabled
        - enabled
      type: ENUM
      values:
        - disabled
        - enabled
public: true
id: mediaPlaybackShuffle
ocfResourceType: x.com.st.mediashuffle
version: 1
```

Media Playback

Allows for the control of the media playback.

222.1 Definition

```
name: Media Playback
status: proposed
attributes:
  level:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
    type: NUMBER
  playbackStatus:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - pause
            - play
            - stop
    type: ENUM
  values:
    - pause
    - play
    - stop
  setter: setPlaybackStatus
  enumCommands:
    - command: play
      value: play
    - command: pause
      value: pause
    - command: stop
      value: stop
  commands:
    setPlaybackStatus:
      arguments:
```

```
- name: status
  required: true
  schema:
    type: string
    enum:
      - pause
      - play
      - stop
    type: ENUM
    values:
      - pause
      - play
      - stop
  play:
    arguments: [ ]
  pause:
    arguments: [ ]
  stop:
    arguments: [ ]
public: true
id: mediaPlayback
ocfResourceType: x.com.st.mediaplayer
version: 1
```

Media Presets

Allows setting a preset from a known list of presets for the media player

223.1 Definition

```
name: Media Presets
status: proposed
attributes:
  presets:
    schema:
      type: object
      properties:
        value:
          type: array
          items:
            $ref: MediaPreset
      type: JSON_OBJECT
commands:
  selectPreset:
    arguments:
      - name: presetId
        required: true
        schema:
          $ref: String
        type: STRING
  playPreset:
    arguments:
      - name: presetId
        required: true
        schema:
          $ref: String
        type: STRING
public: true
id: mediaPresets
version: 1
```

Media Track Control

Allows for the media track control.

224.1 Definition

```
name: Media Track Control
status: proposed
attributes: {
}
commands:
  nextTrack:
    arguments: [
    ]
  previousTrack:
    arguments: [
    ]
public: true
id: mediaTrackControl
ocfResourceType: x.com.st.mediatrackcontrol
version: 1
```

Momentary

Allows for the control of a momentary switch device

225.1 Definition

```
# reviewed 2018-02-22
name: Momentary
status: live
attributes: {
}
commands:
  push:
    arguments: [
]
public: true
id: momentary
ocfResourceType: x.com.st.momentary
version: 1
```

Motion Sensor

Allows for the ability to read motion sensor device states

226.1 Definition

```
# reviewed 2018-01-09
name: Motion Sensor
status: live
attributes:
  motion:
    schema:
      type: object
      properties:
        value:
          $ref: ActivityState
      required: ["value"]
    type: ENUM
    values:
      - active
      - inactive
commands: (
)
public: true
id: motionSensor
ocfResourceType: oic.r.sensor.motion
version: 1
```

Music Player

Allows for control of a music playing device

227.1 Definition

```
# reviewed 2018-02-22
name: Music Player
status: deprecated
attributes:
  level:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
    type: NUMBER
    setter: setLevel
mute:
  schema:
    type: object
    properties:
      value:
        $ref: MuteState
  type: ENUM
  values:
    - muted
    - unmuted
  enumCommands:
    - command: mute
      value: muted
    - command: unmute
      value: unmuted
status:
  schema:
    type: object
    properties:
      value:
        $ref: String
  type: STRING
  actedOnBy:
    - nextTrack
```

```
- pause
- play
- playTrack
- previousTrack
- restoreTrack
- resumeTrack
- setTrack
- stop
trackData:
  schema:
    type: object
    properties:
      value:
        $ref: JsonObject
  type: JSON_OBJECT
actedOnBy:
  - nextTrack
  - pause
  - play
  - playTrack
  - previousTrack
  - restoreTrack
  - resumeTrack
  - setTrack
  - stop
trackDescription:
  schema:
    type: object
    properties:
      value:
        $ref: String
  type: STRING
commands:
  mute:
    arguments: [
    ]
  nextTrack:
    arguments: [
    ]
  pause:
    arguments: [
    ]
  play:
    arguments: [
    ]
  playTrack:
    arguments:
    - name: trackToPlay
      required: true
      schema:
        $ref: String
      type: STRING
  previousTrack:
    arguments: [
    ]
  restoreTrack:
    arguments:
    - name: trackToRestore
```

```
    required: true
    schema:
      $ref: String
    type: STRING
resumeTrack:
  arguments:
    - name: trackToResume
      required: true
      schema:
        $ref: String
      type: STRING
setLevel:
  arguments:
    - name: level
      required: true
      schema:
        $ref: PositiveInteger
      type: NUMBER
setTrack:
  arguments:
    - name: trackToSet
      required: true
      schema:
        $ref: String
      type: STRING
stop:
  arguments: {}
unmute:
  arguments: {}
public: true
id: musicPlayer
version: 1
```

Notification

Allows for displaying notifications on devices that allow notifications to be displayed

228.1 Definition

```
name: Notification
status: live
attributes: {
}
commands:
  deviceNotification:
    arguments:
      - name: notification
        required: true
        schema:
          $ref: String
        type: STRING
public: true
id: notification
version: 1
```

Odor Sensor

Gets the odor sensor reading.

229.1 Definition

```
name: Odor Sensor
status: proposed
attributes:
  odorLevel:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
          required: ["value"]
      type: NUMBER
commands: {}
public: true
id: odorSensor
ocfResourceType: x.com.st.gaslevel
version: 1
```

Outlet

Allows for the control of an outlet device. Deprecated in favor of Switch.

230.1 Definition

```
# reviewed 2018-2-20
name: Outlet
status: deprecated
attributes:
  switch:
    schema:
      type: object
      properties:
        value:
          $ref: SwitchState
      required:
        - value
    type: ENUM
    values:
      - 'off'
      - 'on'
    enumCommands:
      - command: 'on'
        value: 'on'
      - command: 'off'
        value: 'off'
  commands:
    'off':
      arguments: []
    'on':
      arguments: []
public: true
id: outlet
version: 1
```

Oven Mode

Allows for the control of the oven mode.

231.1 Definition

```
name: Oven Mode
status: proposed
attributes:
  ovenMode:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - heating
            - grill
            - warming
            - defrosting
      constraints:
        constraints:
          type: object
          properties:
            values:
              type: array
              items:
                type: string
                enum:
                  - heating
                  - grill
                  - warming
                  - defrosting
          required: ["value"]
    type: ENUM
    values:
      - heating
      - grill
      - warming
      - defrosting
    setter: setOvenMode
commands:
```

```
setOvenMode:
  arguments:
    - name: mode
      required: true
      schema:
        type: string
        enum:
          - heating
          - grill
          - warming
          - defrosting
      type: ENUM
      values:
        - heating
        - grill
        - warming
        - defrosting
  public: true
  id: ovenMode
  ocfResourceType: x.com.st.mode.oven
  version: 1
```

Oven Operating State

Allows for the control of the oven operational state.

232.1 Definition

```
name: Oven Operating State
status: proposed
attributes:
  machineState:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - ready
            - running
            - paused
    type: ENUM
    values:
      - ready
      - running
      - paused
    setter: setMachineState
    actedOnBy:
      - stop
  supportedMachineStates:
    schema:
      type: object
      properties:
        value:
          type: array
          items:
            type: string
            enum:
              - ready
              - running
              - paused
    type: JSON_OBJECT
  ovenJobState:
    schema:
```

```
type: object
properties:
  value:
    type: string
    enum:
      - cleaning
      - cooking
      - cooling
      - draining
      - preheat
      - ready
      - rinsing
type: ENUM
values:
  - cleaning
  - cooking
  - cooling
  - draining
  - preheat
  - ready
  - rinsing
completionTime:
  schema:
    type: object
    properties:
      value:
        $ref: Iso8601Date
    required:
      - value
  type: DATE
operationTime:
  schema:
    type: object
    properties:
      value:
        $ref: PositiveInteger
  type: NUMBER
actedOnBy:
  - stop
progress:
  schema:
    $ref: IntegerPercent
  type: NUMBER
commands:
  setMachineState:
    arguments:
      - name: state
        required: true
    schema:
      type: string
      enum:
        - stop
    type: ENUM
    values:
      - stop
  stop:
    arguments: [
    ]
```

```
public: true
id: ovenOperatingState
ocfResourceType: x.com.st.operationalstate.oven
version: 1
```

Oven Setpoint

Allows for the control of the oven set point.

233.1 Definition

```
name: Oven Setpoint
status: proposed
attributes:
  ovenSetpoint:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
      required: ["value"]
    type: NUMBER
    setter: setOvenSetpoint
commands:
  setOvenSetpoint:
    arguments:
      - name: setpoint
        required: true
        schema:
          $ref: PositiveInteger
        type: NUMBER
public: true
id: ovenSetpoint
ocfResourceType: x.com.st.temperature.oven
version: 1
```

pH Measurement

Read the pH value off of a pH measurement capable device

234.1 Definition

```
# reviewed 2018-2-20
name: pH Measurement
status: live
attributes:
  pH:
    schema:
      type: object
      properties:
        value:
          type: number
          minimum: 0
          maximum: 14
        unit:
          type: string
          enum:
            - pH
          default: pH
      required:
        - value
      type: NUMBER
    commands: (
  )
public: true
id: pHMeasurement
version: 1
```

Polling

Allows for the polling of devices that support it. Deprecated, devices should schedule their own polling using the scheduling API or use the Ping capability.

235.1 Definition

```
# reviewed 2018-2-20
name: Polling
status: deprecated
attributes: {
}
commands:
  poll:
    arguments: {
}
public: true
id: polling
version: 1
```

Power Consumption Report

Allows periodically reporting the energy and power consumption

236.1 Definition

```
name: Power Consumption Report
status: proposed
attributes:
  powerConsumption:
    schema:
      type: object
      properties:
        value:
          $ref: PowerConsumption
          required: ["value"]
      type: JSON_OBJECT
commands: {}

public: true
id: powerConsumptionReport
version: 1
```

Power Meter

Allows for reading the power consumption from devices that report it

237.1 Definition

```
# reviewed 2018-02-20
name: Power Meter
status: live
attributes:
  power:
    schema:
      type: object
      properties:
        value:
          type: number
        unit:
          type: string
          enum:
            - W
          default: W
      required:
        - value
    type: NUMBER
commands: (
)
public: true
id: powerMeter
ocfResourceType: x.com.st.powermeter
version: 1
```

Power Source

Gives the ability to determine the current power source of the device

238.1 Definition

```
name: Power Source
status: live
attributes:
  powerSource:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - battery
            - dc
            - mains
            - unknown
      required: ["value"]
    type: ENUM
    values:
      - battery
      - dc
      - mains
      - unknown
commands: {}
public: true
id: powerSource
version: 1
```

Presence Sensor

The ability to see the current status of a presence sensor device

239.1 Definition

```
# reviewed 2018-01-09
name: Presence Sensor
status: live
attributes:
  presence:
    schema:
      type: object
      properties:
        value:
          $ref: PresenceState
      required: ["value"]
    type: ENUM
    values:
      - not present
      - present
commands: (
)
public: true
id: presenceSensor
ocfResourceType: oic.r.sensor.presence
version: 1
```

Rapid Cooling

Allows for the control of rapid cooling.

240.1 Definition

```
name: Rapid Cooling
status: proposed
attributes:
  rapidCooling:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - 'off'
            - 'on'
          required: ["value"]
      type: ENUM
      values:
        - 'off'
        - 'on'
    setter: setRapidCooling
commands:
  setRapidCooling:
    arguments:
      - name: rapidCooling
        required: true
    schema:
      type: string
      enum:
        - 'off'
        - 'on'
      type: ENUM
      values:
        - 'off'
        - 'on'
public: true
id: rapidCooling
ocfResourceType: x.com.st.rapidcooling
version: 1
```

Refresh

Allow the execution of the refresh command for devices that support it

241.1 Definition

```
# reviewed 2018-2-13
name: Refresh
status: live
attributes: {
}
commands:
  refresh:
    arguments: [
]
public: true
id: refresh
version: 1
```

Refrigeration Setpoint

Allows for the control of the refrigeration set point.

242.1 Definition

```
name: Refrigeration Setpoint
status: proposed
attributes:
  refrigerationSetpoint:
    schema:
      $ref: Temperature
    type: NUMBER
    setter: setRefrigerationSetpoint
commands:
  setRefrigerationSetpoint:
    arguments:
      - name: setpoint
        required: true
        schema:
          $ref: TemperatureValue
        type: NUMBER
public: true
id: refrigerationSetpoint
ocfResourceType: x.com.st.temperature.refrigeration
version: 1
```

Relative Humidity Measurement

Allow reading the relative humidity from devices that support it

243.1 Definition

```
# reviewed 2018-2-13
name: Relative Humidity Measurement
status: live
attributes:
  humidity:
    schema:
      $ref: Percent
    type: NUMBER
commands: (
)
public: true
id: relativeHumidityMeasurement
ocfResourceType: oic.r.humidity
version: 1
```

Relay Switch

Allows for the control of a relay switch device. This is **deprecated** please use switch instead.

244.1 Definition

```
# reviewed 2018-01-11
name: Relay Switch
status: deprecated
attributes:
  switch:
    schema:
      type: object
      properties:
        value:
          $ref: SwitchState
      required: ["value"]
    type: ENUM
    values:
      - 'off'
      - 'on'
    enumCommands:
      - command: 'on'
        value: 'on'
      - command: 'off'
        value: 'off'
  commands:
    'off':
      arguments: [
    ]
    'on':
      arguments: [
    ]
public: true
id: relaySwitch
version: 1
```

Robot Cleaner Cleaning Mode

Allows for the control of the robot cleaner cleaning mode.

245.1 Definition

```
name: Robot Cleaner Cleaning Mode
status: proposed
attributes:
  robotCleanerCleaningMode:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - auto
            - part
            - repeat
            - manual
            - stop
            - map
      required: ["value"]
    type: ENUM
    values:
      - auto
      - part
      - repeat
      - manual
      - stop
      - map
    setter: setRobotCleanerCleaningMode
commands:
  setRobotCleanerCleaningMode:
    arguments:
      - name: mode
        required: true
    schema:
      type: string
      enum:
        - auto
        - part
```

```
    - repeat
    - manual
    - stop
  type: ENUM
  values:
    - auto
    - part
    - repeat
    - manual
    - stop
public: true
id: robotCleanerCleaningMode
ocfResourceType: x.com.st.robot.cleaner.cleaning
version: 1
```

Robot Cleaner Movement

Allows for the control of the robot cleaner movement.

246.1 Definition

```
name: Robot Cleaner Movement
status: proposed
attributes:
  robotCleanerMovement:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - homing
            - idle
            - charging
            - alarm
            - powerOff
            - reserve
            - point
            - after
            - cleaning
          required: ["value"]
      type: ENUM
      values:
        - homing
        - idle
        - charging
        - alarm
        - powerOff
        - reserve
        - point
        - after
        - cleaning
    setter: setRobotCleanerMovement
commands:
  setRobotCleanerMovement:
    arguments:
      - name: mode
```

```
required: true
schema:
  type: string
  enum:
    - homing
  type: ENUM
  values:
    - homing
public: true
id: robotCleanerMovement
ocfResourceType: x.com.st.robot.cleaner.movement
version: 1
```

Robot Cleaner Turbo Mode

Allows for the control of the robot cleaner turbo mode.

247.1 Definition

```
name: Robot Cleaner Turbo Mode
status: proposed
attributes:
  robotCleanerTurboMode:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - 'on'
            - 'off'
            - 'silence'
          required: ["value"]
      type: ENUM
      values:
        - 'on'
        - 'off'
        - 'silence'
    setter: setRobotCleanerTurboMode
commands:
  setRobotCleanerTurboMode:
    arguments:
      - name: mode
        required: true
        schema:
          type: string
          enum:
            - 'on'
            - 'off'
            - 'silence'
        type: ENUM
        values:
          - 'on'
          - 'off'
          - 'silence'
```

```
public: true
id: robotCleanerTurboMode
ocfResourceType: x.com.st.robot.cleaner.turbo
version: 1
```

Sensor

The Sensor capability is a “tagging” capability. It defines no attributes or commands. In SmartThings terms, it represents that a Device has attributes.

248.1 Definition

```
# reviewed 2018-01-11
name: Sensor
status: deprecated
attributes: {
}
commands: {
}
public: true
id: sensor
version: 1
```

Shock Sensor

A Device that senses whether or not there is a shock

249.1 Definition

```
# reviewed 2018-01-11
name: Shock Sensor
status: deprecated
attributes:
  shock:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - clear
            - detected
          required: ["value"]
      type: ENUM
      values:
        - clear
        - detected
    commands: (
    )
public: true
id: shockSensor
version: 1
```

Signal Strength

Gives the ability to read the signal strength of Devices that support it

250.1 Definition

```
# reviewed 2018-2-13
name: Signal Strength
status: live
attributes:
  lqi:
    schema:
      type: object
      properties:
        value:
          type: integer
          minimum: 0
          maximum: 255
      required:
        - value
    type: NUMBER
  rssi:
    schema:
      type: object
      properties:
        value:
          type: number
          minimum: -200
          maximum: 0
        unit:
          type: string
          enum:
            - dBm
          default: dBm
      required:
        - value
    type: NUMBER
commands: (
)
public: true
id: signalStrength
ocfResourceType: x.com.st.signalstrength
version: 1
```

Sleep Sensor

A Device that senses whether or not someone is sleeping

251.1 Definition

```
# reviewed 2018-01-11
name: Sleep Sensor
status: live
attributes:
  sleeping:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - not sleeping # if ever replaced deal with this space (awake)
            - sleeping
          required: ["value"]
      type: ENUM
      values:
        - not sleeping
        - sleeping
    commands: (
    )
public: true
id: sleepSensor
version: 1
```

Smoke Detector

A device that detects the presence or absence of smoke.

252.1 Definition

```
# reviewed 2018-01-09
name: Smoke Detector
status: live
attributes:
  smoke:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - clear
            - detected
            - tested
          required: ["value"]
      type: ENUM
      values:
        - clear
        - detected
        - tested
    commands: (
  )
public: true
id: smokeDetector
ocfResourceType: x.com.st.smokedetector
version: 1
```

Sound Pressure Level

Gets the value of the sound pressure level.

253.1 Definition

```
# reviewed 2018-2-13
name: Sound Pressure Level
status: proposed
attributes:
  soundPressureLevel:
    schema:
      type: object
      properties:
        value:
          type: number
      required:
        - value
    type: NUMBER
commands: (
)
public: true
id: soundPressureLevel
version: 1
```

Sound Sensor

A Device that senses sound

254.1 Definition

```
# reviewed 2018-01-11
name: Sound Sensor
status: live
attributes:
  sound:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - detected
            - not detected
          required: ["value"]
      type: ENUM
      values:
        - detected
        - not detected
    commands: (
    )
public: true
id: soundSensor
ocfResourceType: x.com.st.soundsensor
version: 1
```

Speech Recognition

Gets the spoken phrase string.

255.1 Definition

```
# reviewed 2018-2-13
name: Speech Recognition
status: proposed
attributes:
  phraseSpoken:
    schema:
      type: object
      properties:
        value:
          type: string
          maxLength: 1000
      required:
        - value
    type: STRING
commands: (
)
public: true
id: speechRecognition
version: 1
```

Speech Synthesis

Allows for the control by speech.

256.1 Definition

```
# reviewed 2018-2-13
name: Speech Synthesis
status: proposed
attributes: {
}
commands:
  speak:
    arguments:
      - name: phrase
        required: true
        schema:
          type: string
          maxLength: 1000
    type: STRING
public: true
id: speechSynthesis
version: 1
```

Step Sensor

A Device that works as a step counter

257.1 Definition

```
# reviewed 2018-01-11
name: Step Sensor
status: proposed
attributes:
  goal:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
      required: ["value"]
    type: NUMBER
  steps:
    schema:
      type: object
      properties:
        value:
          $ref: PositiveInteger
      required: ["value"]
    type: NUMBER
commands: {}
public: true
id: stepSensor
version: 1
```

Switch Level

Allows for the control of the level attribute of a light

258.1 Definition

```
# reviewed 2018-01-09 pending decision on rate
name: Switch Level
status: live
attributes:
  level:
    schema:
      $ref: IntegerPercent
    type: NUMBER
    setter: setLevel
commands:
  setLevel:
    arguments:
      - name: level
        schema:
          type: integer
          minimum: 0
          maximum: 100
        type: NUMBER
        required: true
      - name: rate
        schema:
          $ref: PositiveInteger
        type: NUMBER
        required: false
public: true
id: switchLevel
ocfResourceType: oic.r.light.dimming
version: 1
```

Switch

Allows for the control of a switch device

259.1 Definition

```
# reviewed 2018-01-09
name: Switch
status: live
attributes:
  switch:
    schema:
      type: object
      properties:
        value:
          $ref: SwitchState
      required: ["value"]
    type: ENUM
    values:
      - 'off'
      - 'on'
    enumCommands:
      - command: 'on'
        value: 'on'
      - command: 'off'
        value: 'off'
  commands:
    'off':
      arguments: {}
    'on':
      arguments: {}
public: true
id: switch
ocfResourceType: x.com.st.powerswitch
version: 1
```

Tamper Alert

Gets the value of the tamper alert.

260.1 Definition

```
# reviewed 2018-2-13
name: Tamper Alert
status: live
attributes:
  tamper:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - clear
            - detected
      required:
        - value
    type: ENUM
    values:
      - clear
      - detected
commands: (
)
public: true
id: tamperAlert
ocfResourceType: x.com.st.tamperalert
version: 1
```

Temperature Measurement

Get the temperature from a Device that reports current temperature

261.1 Definition

```
# reviewed 2018-01-30
name: Temperature Measurement
status: live
attributes:
  temperature:
    schema:
      type: object
      properties:
        value:
          $ref: TemperatureValue
        unit:
          $ref: TemperatureUnit
      required: ["value", "unit"]
    type: NUMBER
commands: (
)
public: true
id: temperatureMeasurement
ocfResourceType: x.com.st.temperature.measured
version: 1
```

Thermostat Cooling Setpoint

Allows for setting the cooling setpoint on a thermostat

262.1 Definition

```
# reviewed 2018-01-30
name: Thermostat Cooling Setpoint
status: live
attributes:
  coolingSetpoint:
    schema:
      $ref: Temperature
    type: NUMBER
    setter: setCoolingSetpoint
commands:
  setCoolingSetpoint:
    arguments:
      - name: setpoint
        required: true
        schema:
          $ref: TemperatureValue
        type: NUMBER
public: true
id: thermostatCoolingSetpoint
ocfResourceType: x.com.st.temperature.cooling
version: 1
```

Thermostat Fan Mode

263.1 Definition

```
# reviewed 2018-01-30
name: Thermostat Fan Mode
status: live
attributes:
  thermostatFanMode:
    schema:
      type: object
      properties:
        value:
          $ref: ThermostatFanMode
      required:
        - value
    type: ENUM
    values:
      - auto
      - circulate
      - followschedule
      - 'on'
    setter: setThermostatFanMode
    enumCommands:
      - command: fanAuto
        value: auto
      - command: fanCirculate
        value: circulate
      - command: fanOn
        value: 'on'
    supportedThermostatFanModes:
      schema:
        type: object
        properties:
          value:
            type: array
            items:
              $ref: ThermostatFanMode

    type: JSON_OBJECT
commands:
  fanAuto:
    arguments: |
```

```
    |
  fanCirculate:
    arguments: |
      |
  fanOn:
    arguments: |
      |
  setThermostatFanMode:
    arguments:
      - name: mode
        required: true
        schema:
          $ref: ThermostatFanMode
        type: ENUM
        values:
          - auto
          - circulate
          - followschedule
          - 'on'
  public: true
  id: thermostatFanMode
  ocfResourceType: x.com.st.mode.fan.thermostat
  version: 1
```

Thermostat Heating Setpoint

Allows for setting the heating setpoint on a thermostat

264.1 Definition

```
# reviewed 2018-01-30
name: Thermostat Heating Setpoint
status: live
attributes:
  heatingSetpoint:
    schema:
      $ref: Temperature
    type: NUMBER
    setter: setHeatingSetpoint
commands:
  setHeatingSetpoint:
    arguments:
      - name: setpoint
        required: true
        schema:
          $ref: TemperatureValue
        type: NUMBER
public: true
id: thermostatHeatingSetpoint
ocfResourceType: x.com.st.temperature.heating
version: 1
```

Thermostat Mode

265.1 Definition

```
# reviewed 2018-01-30
name: Thermostat Mode
status: live
attributes:
  thermostatMode:
    schema:
      type: object
      properties:
        value:
          $ref: ThermostatMode
      required:
        - value
    type: ENUM
  values:
    - auto
    - eco
    - rush hour
    - cool
    - emergency heat
    - heat
    - 'off'
  setter: setThermostatMode
  enumCommands:
    - command: auto
      value: auto
    - command: cool
      value: cool
    - command: emergencyHeat
      value: emergency heat
    - command: heat
      value: heat
    - command: 'off'
      value: 'off'
  supportedThermostatModes:
    schema:
      type: object
      properties:
        value:
          type: array
```

```
    items:
      $ref: ThermostatMode
    type: JSON_OBJECT
  commands:
    auto:
      arguments: {}
    cool:
      arguments: {}
    emergencyHeat:
      arguments: {}
    heat:
      arguments: {}
    'off':
      arguments: {}
    setThermostatMode:
      arguments:
        - name: mode
          required: true
          schema:
            $ref: ThermostatMode
      type: ENUM
      values:
        - auto
        - eco
        - rush hour
        - cool
        - emergency heat
        - heat
        - 'off'
  public: true
  id: thermostatMode
  ocfResourceType: x.com.st.mode.thermostat
  version: 1
```

Thermostat Operating State

Gives the ability to see the current state that the thermostat is operating in

266.1 Definition

```
# reviewed 2018-01-30
name: Thermostat Operating State
status: live
attributes:
  thermostatOperatingState:
    schema:
      type: object
      properties:
        value:
          $ref: ThermostatOperatingState
      required:
        - value
    type: ENUM
    values:
      - cooling
      - fan only
      - heating
      - idle
      - pending cool
      - pending heat
      - vent economizer
  commands: {}
public: true
id: thermostatOperatingState
ocfResourceType: x.com.st.operationalstate.thermostat
version: 1
```

Thermostat Setpoint

Gives the ability to read the current setpoint on a thermostat

267.1 Definition

```
# reviewed 2018-01-30
name: Thermostat Setpoint
status: deprecated
attributes:
  thermostatSetpoint:
    schema:
      $ref: Temperature
    type: NUMBER
commands: (
)
public: true
id: thermostatSetpoint
ocfResourceType: x.com.st.temperature.setpoint
version: 1
```

Thermostat

Allows for the control of a thermostat device

268.1 Definition

```
# reviewed 2018-01-30
name: Thermostat
status: deprecated
attributes:
  coolingSetpoint:
    schema:
      $ref: Temperature
    type: NUMBER
    setter: setCoolingSetpoint
  coolingSetpointRange:
    schema:
      type: object
      properties:
        value:
          type: array
          items:
            - $ref: TemperatureValue
            - $ref: TemperatureValue
          minItems: 2
          maxItems: 2
      required:
        - value
    type: VECTOR3
  heatingSetpoint:
    schema:
      $ref: Temperature
    type: NUMBER
    setter: setHeatingSetpoint
  heatingSetpointRange:
    schema:
      type: object
      properties:
        value:
          type: array
          items:
            - $ref: TemperatureValue
```

```
    - $ref: TemperatureValue
    minItems: 2
    maxItems: 2
  required:
  - value
  type: VECTOR3
  schedule:
    schema:
      type: object
      properties:
        value:
          $ref: JsonObject
      required:
      - value
    type: JSON_OBJECT
    setter: setSchedule
  temperature:
    schema:
      $ref: Temperature
    type: NUMBER
  thermostatFanMode:
    schema:
      type: object
      properties:
        value:
          $ref: ThermostatFanMode
      required:
      - value
    type: ENUM
  values:
  - auto
  - circulate
  - followschedule
  - 'on'
  setter: setThermostatFanMode
  enumCommands:
  - command: fanAuto
    value: auto
  - command: fanCirculate
    value: circulate
  - command: fanOn
    value: 'on'
  supportedThermostatFanModes:
    schema:
      type: object
      properties:
        value:
          type: array
          items:
            $ref: ThermostatFanMode
    type: JSON_OBJECT
  thermostatMode:
    schema:
      type: object
      properties:
        value:
          $ref: ThermostatMode
      required:
```

```
- value
type: ENUM
values:
  - auto
  - eco
  - rush hour
  - cool
  - emergency heat
  - heat
  - 'off'
setter: setThermostatMode
enumCommands:
  - command: auto
    value: auto
  - command: cool
    value: cool
  - command: emergencyHeat
    value: emergency heat
  - command: heat
    value: heat
  - command: 'off'
    value: 'off'
supportedThermostatModes:
  schema:
    type: object
    properties:
      value:
        type: array
        items:
          $ref: ThermostatMode
  type: JSON_OBJECT
thermostatOperatingState:
  schema:
    type: object
    properties:
      value:
        $ref: ThermostatOperatingState
    required:
      - value
  type: ENUM
values:
  - cooling
  - fan only
  - heating
  - idle
  - pending cool
  - pending heat
  - vent economizer
thermostatSetpoint:
  schema:
    $ref: Temperature
  type: NUMBER
thermostatSetpointRange:
  schema:
    type: object
    properties:
      value:
        type: array
```

```
    items:
      - $ref: TemperatureValue
      - $ref: TemperatureValue
    minItems: 2
    maxItems: 2
  type: VECTOR3
commands:
  auto:
    arguments: {}
  cool:
    arguments: {}
  emergencyHeat:
    arguments: {}
  fanAuto:
    arguments: {}
  fanCirculate:
    arguments: {}
  fanOn:
    arguments: {}
  heat:
    arguments: {}
  'off':
    arguments: {}
  setCoolingSetpoint:
    arguments:
      - name: setpoint
        required: true
        schema:
          $ref: TemperatureValue
    type: NUMBER
  setHeatingSetpoint:
    arguments:
      - name: setpoint
        required: true
        schema:
          $ref: TemperatureValue
    type: NUMBER
  setSchedule:
    arguments:
      - name: schedule
        required: true
        schema:
          $ref: JsonObject
    type: JSON_OBJECT
  setThermostatFanMode:
    arguments:
      - name: fanmode
        required: true
        schema:
          $ref: ThermostatFanMode
```

```
type: ENUM
values:
  - auto
  - circulate
  - followschedule
  - 'on'
setThermostatMode:
arguments:
  - name: mode
    required: true
    schema:
      $ref: ThermostatMode
type: ENUM
values:
  - auto
  - eco
  - rush hour
  - cool
  - emergency heat
  - heat
  - 'off'
public: true
id: thermostat
version: 1
```

Three Axis

Gives the three axis coordinates for devices that support it

269.1 Definition

```
# reviewed 2018-2-13
name: Three Axis
status: live
attributes:
  threeAxis:
    schema:
      type: object
      properties:
        value:
          type: array
          items:
            type: integer
            minimum: -10000
            maximum: 10000
          minItems: 3
          maxItems: 3
        unit:
          type: string
          enum:
            - mG
          default: mG
      required:
        - value
    type: VECTOR3
commands: (
)
public: true
id: threeAxis
version: 1
```

Timed Session

Allows for the control of the timed session.

270.1 Definition

```
# reviewed 2018-2-13
name: Timed Session
status: proposed
attributes:
  sessionStatus:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - canceled
            - paused
            - running
            - stopped
      constraints:
        type: object
        properties:
          values:
            type: array
            items:
              type: string
              enum:
                - canceled
                - paused
                - running
                - stopped
    required:
      - value
  type: ENUM
  values:
    - canceled
    - paused
    - running
    - stopped
  enumCommands:
```

```
- command: cancel
  value: canceled
- command: pause
  value: paused
- command: start
  value: running
- command: stop
  value: stopped
completionTime:
  schema:
    type: object
    properties:
      value:
        $ref: Iso8601Date
    required:
      - value
  type: DATE
  setter: setCompletionTime
commands:
  cancel:
    arguments: []
  pause:
    arguments: []
  setCompletionTime:
    arguments:
      - name: completionTime
        required: true
    schema:
      $ref: Iso8601Date
    type: DATE
  start:
    arguments: []
  stop:
    arguments: []
public: true
id: timedSession
version: 1
```

Tone

Allows for the control of a device that can make an audible tone

271.1 Definition

```
# reviewed 2018-02-15
name: Tone
status: live
attributes: {
}
commands:
  beep:
    arguments: [
]
public: true
id: tone
ocfResourceType: x.com.st.tone
version: 1
```

Touch Sensor

Gives the ability to get the touched status for devices that are touch sensitive. This has been **deprecated** in favor of the button capability

272.1 Definition

```
# reviewed 2018-01-11
name: Touch Sensor
status: deprecated
attributes:
  touch:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - touched
      type: ENUM
    values:
      - touched
commands: {}
public: true
id: touchSensor
version: 1
```

Tv Channel

Allows for the control of the TV channel.

273.1 Definition

```
name: Tv Channel
status: proposed
attributes:
  tvChannel:
    schema:
      type: object
      properties:
        value:
          $ref: String
    type: STRING
    setter: setTvChannel
    actedOnBy:
      - channelDown
      - channelUp
commands:
  setTvChannel:
    arguments:
      - name: channel
        required: true
        schema:
          $ref: String
        type: STRING
  channelUp:
    arguments: []
  channelDown:
    arguments: []
public: true
id: tvChannel
ocfResourceType: x.com.st.tvchannel
version: 1
```

Ultraviolet Index

Gives the ability to get the ultraviolet index from devices that report it

274.1 Definition

```
# reviewed 2018-02-15
name: Ultraviolet Index
status: live
attributes:
  ultravioletIndex:
    schema:
      type: object
      properties:
        value:
          type: number
          minimum: 0
          maximum: 255
      required: ["value"]
    type: NUMBER
commands: (
)
public: true
id: ultravioletIndex
version: 1
```

Valve

Allows for the control of a valve device

275.1 Definition

```
# reviewed 2018-02-15
name: Valve
status: live
attributes:
  valve:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - closed
            - open
          required: ["value"]
      type: ENUM
      values:
        - closed
        - open
      enumCommands:
        - command: close
          value: closed
        - command: open
          value: open
    commands:
      close:
        arguments: []
      open:
        arguments: []
    public: true
id: valve
ocfResourceType: x.com.st.valve
version: 1
```

Video Clips

Video clip capture

276.1 Definition

```
name: Video Clips
status: proposed
attributes:
  videoClip:
    schema:
      type: object
      properties:
        value:
          $ref: VideoClip
      required:
        - value
    type: JSON_OBJECT
  actedOnBy:
    - captureClip
commands:
  captureClip:
    arguments:
      - name: duration
        required: true
        schema:
          $ref: PositiveInteger
        type: NUMBER
      - name: preFetch
        required: true
        schema:
          $ref: PositiveInteger
        type: NUMBER
public: true
id: videoClips
version: 1
```

Video Stream

Allows for the control of the video stream.

277.1 Definition

```
name: Video Stream
status: proposed
attributes:
  stream:
    schema:
      type: object
      properties:
        value:
          $ref: JsonObject
      required:
        - value
      type: JSON_OBJECT
    actedOnBy:
      - startStream
      - stopStream
commands:
  startStream:
    arguments: {}
  stopStream:
    arguments: {}
public: true
id: videoStream
ocfResourceType: x.com.st.videostream
version: 1
```

Voltage Measurement

Get the value of voltage measured from devices that support it

278.1 Definition

```
# reviewed 2018-02-15
name: Voltage Measurement
status: live
attributes:
  voltage:
    schema:
      type: object
      properties:
        value:
          $ref: Number
        unit:
          type: string
          enum:
            - V
          default: V
      required:
        - value
    type: NUMBER
commands: (
)
public: true
id: voltageMeasurement
version: 1
```

Washer Mode

Allows for the control of the washer mode.

279.1 Definition

```
name: Washer Mode
status: proposed
attributes:
  washerMode:
    schema:
      type: object
      properties:
        value:
          $ref: WasherMode
      required: ["value"]
    type: ENUM
    values:
      - regular
      - heavy
      - rinse
      - spinDry
    setter: setWasherMode
commands:
  setWasherMode:
    arguments:
      - name: mode
        required: true
    schema:
      $ref: WasherMode
    type: ENUM
    values:
      - regular
      - heavy
      - rinse
      - spinDry
public: true
id: washerMode
ocfResourceType: x.com.st.mode.washer
version: 1
```

Washer Operating State

Allows for the control of the washer operational state.

280.1 Definition

```
name: Washer Operating State
status: proposed
attributes:
  machineState:
    schema:
      type: object
      properties:
        value:
          $ref: MachineState
      required: ["value"]
    type: ENUM
    values:
      - pause
      - run
      - stop
    setter: setMachineState
  supportedMachineStates:
    schema:
      type: object
      properties:
        value:
          type: array
          items:
            $ref: MachineState
    type: JSON_OBJECT
  washerJobState:
    schema:
      type: object
      properties:
        value:
          type: string
          enum:
            - airWash
            - cooling
            - delayWash
            - drying
```

```
    - finish
    - none
    - preWash
    - rinse
    - spin
    - wash
    - weightSensing
    - wrinklePrevent
  constraints:
    type: object
  properties:
    values:
      type: array
      items:
        type: string
        enum:
          - airWash
          - cooling
          - delayWash
          - drying
          - finish
          - none
          - preWash
          - rinse
          - spin
          - wash
          - weightSensing
          - wrinklePrevent
  required: ["value"]
type: ENUM
values:
  - airWash
  - cooling
  - delayWash
  - drying
  - finish
  - none
  - preWash
  - rinse
  - spin
  - wash
  - weightSensing
  - wrinklePrevent
completionTime:
  schema:
    type: object
    properties:
      value:
        $ref: Iso8601Date
    required:
      - value
  type: DATE
commands:
  setMachineState:
    arguments:
      - name: state
        required: true
    schema:
```



```
    $ref: MachineState
  type: ENUM
  values:
    - pause
    - run
    - stop
public: true
id: washerOperatingState
ocfResourceType: x.com.st.operationalstate.washer
version: 1
```

Water Sensor

Get the status off of a water sensor device

281.1 Definition

```
# reviewed 2018-01-09
name: Water Sensor
status: live
attributes:
  water:
    schema:
      type: object
      properties:
        value:
          $ref: MoistureState
      required: ["value"]
    type: ENUM
    values:
      - dry
      - wet
commands: (
)
public: true
id: waterSensor
ocfResourceType: oic.r.sensor.water
version: 1
```

Window Shade

Allows for the control of the window shade.

282.1 Definition

```
# reviewed 2018-02-15
name: Window Shade
status: proposed
attributes:
  windowShade:
    schema:
      type: object
      properties:
        value:
          $ref: OpenableState
        constraints:
          type: object
          properties:
            values:
              type: array
              items:
                $ref: OpenableState
      required:
        - value
    type: ENUM
    values:
      - closed
      - closing
      - open
      - opening
      - partially open
      - unknown
    enumCommands:
      - command: close
        value: closed
      - command: open
        value: open
    actedOnBy:
      - presetPosition
commands:
  close:
```

```
arguments: {  
  |  
  open:  
    arguments: {  
      |  
      presetPosition:  
        arguments: {  
          |  
        }  
      }  
    }  
  }  
public: true  
id: windowShade  
version: 1
```

Part XX

API Documentation

This is where you can find API-level documentation for the various objects available in your SmartApps and Device Handlers.

How to read the docs

283.1 Objects

SmartThings objects are rarely created directly by SmartApp or Device Handler developers. Instead, various objects are already created and available in your applications.

You will rarely see constructor documentation for this reason. Each object will contain a summary at the top of the document that discusses some of the common ways to get a reference to the object.

Also worth noting is that some of the “objects” documented are not really objects at all. A SmartApp is not an object, in the strict sense of the word, for example (neither is a Device Handler). But each running execution of a SmartApp or Device Handler has available to it many methods and properties. For convenience, we have organized the methods available to SmartApps and Device Handlers into the SmartApp or Device Handler API documentation.

283.2 Object wrappers

You may notice in various log messages or error messages that the objects are actually wrapper objects. For example, `Event` is actually an instance of `EventWrapper`. We have wrapped many of our core objects with wrapper objects to protect access to the underlying system object.

This should be transparent to developers, since these objects cannot be instantiated directly. The underlying wrapper class may even change at some point (the supported APIs should not, without notice).

But, should you be confused about messages in the log, this is why.

283.3 Dynamic methods

The Groovy programming language offers a powerful feature called [Metaprogramming](#) that (among other things) allows for Groovy programs to be written in a way that *methods can be created dynamically at run time*.

SmartThings makes use of this powerful feature in a few ways. For example, you can get a reference to a Device configured in a SmartApp preference by simply referencing the name of the device configured in the preference. Another example is getting various Attribute values for a Device by invoking a method in the form `<someDevice>.current<AttributeName>`.

This powerful feature can make documenting all available methods difficult, since methods may not exist until runtime. For any dynamic methods, the method or property will be enclosed in `<>`, and a description and example will be given.

283.4 Conventions

All methods are listed in alphabetical order, with the exception of SmartApp and Device Handler methods that are expected to be defined by SmartApps and Device Handlers - those will be listed first.

Note: Groovy follows the JavaBean convention, and adds some syntactic sugar on top. Any zero-arg getter can be retrieved via property access directly. For example, `getName()` could be invoked as `name`. You'll see this shortcut syntax often in Groovy and SmartThings.

Some methods may have many signatures. For example, the `schedule` method available to SmartApps can be called with a variety of arguments. We have documented all forms in one location (`schedule()`). All supported signatures will be listed, as well as all parameters for the various signatures.

Optional parameters will be listed inside brackets (`[]`) in the method signature.

Code examples may not be executable as-is. Since SmartApps and Device Handlers execute in response to various schedules or Events, and rely upon having other metadata defined, the examples have been written with brevity in mind. The code samples may need to be defined inside an event handler or otherwise executable code block to fully function.

When appropriate, we have included various tips or warnings. In cases where an API is not adequately documented currently, we have called attention to that. We plan to add the supporting documentation soon!

API Contents

284.1 SmartApp

A SmartApp is a Groovy-based program that allows developers to create automations for users to tap into the capabilities of their devices.

They are created through the “New SmartApp” action in the IDE. There is no “class” for a SmartApp per se, but there are various methods and properties available to SmartApps that are documented below.

When a SmartApp executes, it executes in the context of a certain installation instance. That is, a user installs a SmartApp on their mobile application, and configures it with devices or rules unique to them. A SmartApp is not continuously running; it is executed in response to various schedules or subscribed-to Events.

The following methods should be defined by all SmartApps. They are called by the SmartThings platform at various points in the SmartApp lifecycle.

284.1.1 installed()

Note: This method is expected to be defined by SmartApps.

Called when an instance of the app is installed. Typically subscribes to Events from the configured devices and creates any scheduled jobs.

Signature: void installed()

Returns: void

Example:

```
def installed() {
    log.debug "installed with settings: $settings"

    // subscribe to events, create scheduled jobs.
}
```

284.1.2 updated()

Note: This method is expected to be defined by SmartApps.

Called when the preferences of an installed app are updated. Typically unsubscribes and re-subscribes to Events from the configured devices and unschedules/reschedules jobs.

Signature: void uninstalled()

Returns: void

Example:

```
def updated() {
    unsubscribe()
    // resubscribe to device events, create scheduled jobs
}
```

284.1.3 uninstalled()

Note: This method may be defined by SmartApps.

Called, if declared, when an app is uninstalled. Does not need to be declared unless you have some external cleanup to do. subscriptions and scheduled jobs are automatically removed when an app is uninstalled, so you don't need to do that here.

Signature: void uninstalled()

Returns: void

Example:

```
def uninstalled() {
    // external cleanup. No need to unsubscribe or remove scheduled jobs
}
```

The following methods and attributes are available to call in a SmartApp:

284.1.4 <device or capability preference name>

A reference to the device or devices selected during app installation or update.

Returns: *Device* (page 1003) or a list of Devices - the Device with the given preference name, or a list of Devices if `multiple:true` is specified in the preferences.

Example:

```
preferences {
    ...
    input "theswitch", "capability.switch"
    input "theswitches", "capability.switch", multiple:true
    ...
}
```

```

}

...
// the name of the preference becomes the reference for the Device object
theswitch.on()
theswitch.off()

// multiple:true means we get a list of devices
theswitches.each {log.debug "Current switch value: ${it.currentSwitch}" }

// we can still call methods directly on the list; it will apply the method to each device:
theswitches.on() // turn all switches on

```

284.1.5 <number or decimal preference name>

A reference to the value entered for a number or decimal input preference.

Returns: `BigDecimal` - the value entered for a number or decimal input preference.

Example:

```

preferences {
    ...
    input "num1", "number"
    input "dec1", "decimal"
    ...
}

...
// preference name is a reference to a BigDecimal that is the value the user entered.
log.debug "num1: $num1" //=> value user entered for num1 preference
log.debug "dec1: $dec1" //=> value user entered for dec1 preference
...

```

284.1.6 <text, mode, or time preference name>

A reference to the value entered for a text, mode, or time input type.

The following table explains the value and format returned for the various input types:

Input Type	Return Value
text	<code>String</code> - the value entered as text
mode	<code>String</code> - the name of the mode selected
time	<code>String</code> - the full date string in the format of “yyyy-MM-dd’T’HH:mm:ss.SSSZ”

Example:

```

preferences {
    ...
    input "mytext", "text"
    input "mymode", "mode"
    input "mytime", "time"
    ...
}

```

```
}  
  
log.debug "mytext: $mytext "  
log.debug "mymode: $mymode "  
log.debug "mytime: $mytime "  
  
// time is in format compatible with most scheduling APIs.  
// we can pass the value directly to the APIs that accept a date string:  
runOnce(mytime, someHandlerMethod)  
schedule(myTime, someHandlerMethod)
```

284.1.7 addChildApp()

Adds a child app to a SmartApp.

Warning: A SmartApp may have a maximum of 500 child SmartApps and devices, combined.

Signature: `InstalledSmartApp addChildApp(String namespace, String smartAppVersionName, String label, Map properties)`

Throws: `IllegalArgumentException` - If a label was not supplied `NotFoundException` - If the given SmartApp name was not found in the given Namespace.

Parameters: `String namespace` - the namespace of the child SmartApp

`String smartAppVersionName` - the name of the SmartApp

`String label` - a label to give the child app

`Map properties` (*optional*) - A map with SmartApp properties for the child app.

Returns: `InstalledSmartApp` (page 1031) - The `InstalledSmartAppWrapper` instance that represents the child SmartApp that was created.

Throws: `IllegalArgumentException` - If the label is not provided.

`NotFoundException` - If the SmartApp cannot be found.

`SizeLimitExceededException` - If this SmartApp already has the maximum number of children allowed (500).

284.1.8 addChildDevice()

Adds a child device to a SmartApp. An example use is in Service Manager SmartApps.

Warning: A parent may have at most 500 children.

Signature: `DeviceWrapper addChildDevice(String typeName, String deviceNetworkId, hubId, Map properties)`

`DeviceWrapper addChildDevice(String namespace, String typeName, String deviceNetworkId, hubId, Map properties)`

Parameters: `String namespace` - the namespace for the device. Defaults to `installedSmartApp.smartAppVersionDTO.smartAppDTO.namespace`

`String typeName` - the device type name

`String deviceNetworkId` - the device network id of the device

`hubId` - (*optional*) The Hub id. Defaults to null

`Map properties` (*optional*) - A map with device properties.

Returns: *Device* (page 1003) - The device that was created.

Throws: *UnknownDeviceTypeException* - If a Device Handler with the specified name and namespace is not found.

IllegalArgumentException - If the `deviceNetworkId` is not specified.

SizeLimitExceededException - If this SmartApp already has the maximum number of children allowed (500).

284.1.9 apiServerUrl()

Returns the URL of the server where this SmartApp can be reached for API calls, along with the specified path appended to it. Use this instead of hard-coding a URL to ensure that the correct server URL for this installed instance is returned.

Signature: `String apiServerUrl(String path)`

Parameters: `String path` - the path to append to the URL

Returns: The URL of the server for this installed instance of the SmartApp.

Example:

```
// logs <server url>/my/path
log.debug "apiServerUrl: ${apiServerUrl("/my/path")}"

// The leading "/" will be added if you don't specify it
// logs <server url>/my/path
log.debug "apiServerUrl: ${apiServerUrl("my/path")}"
```

284.1.10 atomicState

A map of name/value pairs that SmartApp can use to save and retrieve data across SmartApp executions. This is similar to *state* (page 941), but will immediately write and read from the backing data store. Prefer using *state* over *atomicState* when possible.

Signature: `Map atomicState`

Returns: `Map` - a map of name/value pairs.

```
atomicState.count = 0
atomicState.count = atomicState.count + 1

log.debug "atomicState.count: ${atomicState.count}"

// use array notation if you wish
log.debug "atomicState['count']: ${atomicState['count']}"

// you can store lists and maps to make more interesting structures
```

```
atomicState.listOfMaps = [[key1: "vall", bool1: true],  
                          [otherKey: ["string1", "string2"]]]
```

284.1.11 canSchedule()

Returns true if the SmartApp is able to schedule jobs. SmartApps are limited to 6 pending scheduled executions.

Signature: Boolean canSchedule()

Returns: Boolean - true if additional jobs can be scheduled, false otherwise.

Example:

```
log.debug "Can schedule? ${canSchedule()}"
```

284.1.12 createAccessToken()

Creates an access token for this installed SmartApp. This token is intended to be used by third-party services that need to communicate with SmartThings during the *OAuth installation flow of cloud-connected devices* (page 546).

The created token will then be available in `state.accessToken`.

Signature: def createAccessToken()

Returns: May return the access token itself, though this is not guaranteed (the token will be available in `state.accessToken`).

Example:

```
// Check to see if SmartApp has its own access token and create one if not.  
if(!state.accessToken) {  
    // the createAccessToken() method will store the access token in state.accessToken  
    createAccessToken()  
}  
  
// Use token to allow third-party to communicate with SmartApp during setup  
  
// Revoke the token once the third-party no longer needs it (after setup)  
revokeAccessToken()
```

See also:

- *Building the Service Manager* (page 546)
 - *revokeAccessToken()* (page 928)
-

284.1.13 findAllChildAppsByName()

Finds all child SmartApps matching the specified name. This includes child SmartApps that have both “complete” and “incomplete” *installation states* (page 1033).

Signature: List<InstalledSmartApp> findAllChildAppsByName(String namespace, String name)

Parameters: `String` name - the name of the SmartApp to find.

Returns: A list of *InstalledSmartApp* (page 1031), or an empty list if none are found.

Example:

```
def children = findAllChildAppsByName("My Child App")
log.debug "found ${children.size()} child apps"

children.each { child ->
    log.debug "child app ${child.id} has installation state ${child.installationState}"
}
```

284.1.14 findAllChildAppsByNamespaceAndName()

Finds all child SmartApps matching the specified namespace and name. This includes child SmartApps that have both “complete” and “incomplete” *installation states* (page 1033).

Signature: `List<InstalledSmartApp> findAllChildAppsByNamespaceAndName(String namespace, String name)`

Parameters: `String` namespace - the namespace of the SmartApp to find.

`String` name - the name of the SmartApp to find.

Returns: A list of *InstalledSmartApp* (page 1031), or an empty list if none are found.

Example:

```
def children = findAllChildAppsByNamespaceAndName("somenamespace", "My Child App")
log.debug "found ${children.size()} child apps"

children.each { child ->
    log.debug "child app ${child.id} has installation state ${child.installationState}"
}
```

284.1.15 findChildAppByName()

Finds a child SmartApp matching the specified name. This includes child SmartApps that have both “complete” and “incomplete” *installation states* (page 1033).

Signature: `def findChildAppByName(String appName)`

Parameters: `String` appName - the name of the SmartApp to find.

Returns: A *InstalledSmartApp* (page 1031) if a child app is found that matches the specified name; `null` if no child app that matches the name is found. If there are multiple child apps that match the specified name, only the first one found will be returned.

Example:

```
def child = findChildAppByName("My Child App")
log.debug "child app id ${child?.id} has installation state ${child.installationState}"
```

284.1.16 findChildAppByNamespaceAndName()

Finds a child SmartApp matching the specified namespace and name. This includes child SmartApps that have both “complete” and “incomplete” *installation states* (page 1033).

Signature: `def findChildAppsByNamespaceAndName(String namespace, String name)`

Parameters: `String namespace` - the namespace of the SmartApp to find.

`String name` - the name of the SmartApp to find.

Returns: A *InstalledSmartApp* (page 1031), or null if no child app is found. If multiple child apps are found that match the namespace and name, the first one will be returned.

Example:

```
def child = findChildAppByNamespaceAndName("somenamespace", "My Child App")
log.debug "child app id ${child?.id} has installation state ${child.installationState}"
```

284.1.17 getAllChildApps()

Gets a list of child apps associated with this SmartApp. This includes child SmartApps that have both “complete” and “incomplete” *installation states* (page 1033).

Signature: `List<InstalledSmartApp> getAllChildApps()`

Returns: `List<InstalledSmartApp>` (page 1031) - A list of child SmartApps

Example:

```
def childApps = app.getAllChildApps()
log.debug "This app has ${childApps.size()} child apps"

childApps.each { child ->
    log.debug "child app with id ${child.id} has installation state ${child.installationState}"
}
```

284.1.18 getChildApps()

Gets a list of child apps associated with this SmartApp. This only includes child SmartApps that have an *installation state* (page 1033) of “complete”.

Signature: `List<InstalledSmartApp> getChildApps()`

Returns: `List<InstalledSmartApp>` (page 1031) - A list of child SmartApps

Example:

```
def childApps = getChildApps()

// Update the label for all child apps
childApps.each {
    if (!it.label?.startsWith(app.name)) {
        it.updateLabel("${app.name}/${it.label}")
    }
}
```

284.1.19 deleteChildDevice()

Deletes the child device with the specified device network id.

Signature: `void deleteChildDevice(String deviceNetworkId)`

Throws: `NotFoundException`

Parameters: `String deviceNetworkId` - the device network id of the device

Returns: `void`

284.1.20 getAllChildDevices()

Returns a list of all child devices, including virtual devices. This is a wrapper for `getChildDevices(true)`.

Signature: `List getChildDevices()`

Returns: `List` - a list of all child devices.

284.1.21 getApiServerUrl()

Returns the URL of the server where this SmartApp can be reached for API calls. Use this instead of hard-coding a URL to ensure that the correct server URL for this installed instance is returned.

Signature: `String getApiServerUrl()`

Returns: `String` - the URL of the server where this SmartApp can be reached.

284.1.22 getChildDevice()

Returns a device based upon the specified device network id. This is mostly used in Service Manager SmartApps.

Signature: `DeviceWrapper getChildDevice(String deviceNetworkId)`

Parameters: `String deviceNetworkId` - the device network id of the device

Returns: `DeviceWrapper` - The device found with the given device network ID.

284.1.23 getChildDevices()

Returns a list of all child devices. An example use would be in Service Manager SmartApps.

Signature: `List getChildDevices(Boolean includeVirtualDevices)`

Parameters: `Boolean true` if the returned list should contain virtual devices. Defaults to `false`. (*optional*)

Returns: `List` - A list of all devices found.

284.1.24 getColorUtil()

Returns the *ColorUtilities* (page 1001) object.

Signature: `ColorUtilities getColorUtil()`

Returns: *ColorUtilities* (page 1001)

284.1.25 getLocation()

The *Location* (page 1036) into which this SmartApp has been installed.

Signature: `Location getLocation()`

Returns: *Location* (page 1036) - The Location into which this SmartApp has been installed.

284.1.26 getSunriseAndSunset()

Gets a map containing the local sunrise and sunset times.

Signature: `Map getSunriseAndSunset([Map options])`

Parameters:

Map options (optional)

The supported options are:

Option	Description
zipCode	<i>String</i> - the zip code to use for determining the times. If not specified then the coordinates of the Hub location are used.
locationString	<i>String</i> - any location string supported by the Weather Underground APIs. If not specified then the coordinates of the Hub Location are used
sunriseOffset	<i>String</i> - adjust the sunrise time by this amount. See <i>timeOffset()</i> (page 944) for supported formats
sunsetOffset	<i>String</i> - adjust the sunset time by this amount. See <i>timeOffset()</i> (page 944) for supported formats

Returns: Map - A Map containing the local sunrise and sunset times as Date objects: [sunrise: Date, sunset: Date]

Example:

```
def noParams = getSunriseAndSunset ()
def beverlyHills = getSunriseAndSunset (zipCode: "90210")
def thirtyMinsBeforeSunset = getSunriseAndSunset (sunsetOffset: "-00:30")

log.debug "sunrise with no parameters: ${noParams.sunrise}"
log.debug "sunset with no parameters: ${noParams.sunset}"
log.debug "sunrise and sunset in 90210: $beverlyHills"
log.debug "thirty minutes before sunset at current Location: ${thirtyMinsBeforeSunset.sunset}"
```

284.1.27 getTwcConditions()

Note: If you are considering the development of an application that makes extensive use of weather data, you should consider gaining direct access to APIs from a weather data provider.

Get the current weather conditions.

Signature: def getTwcConditions(String locationString = null)

Parameters: String locationString - Optional. Must be a 5 digit US zip code or a latitude, longitude string (e.g., "38.25,-76.45"). If not specified, the method will use the latitude and longitude of the Location as set in the SmartThings mobile app.

Example Response:

```
cloudCeiling: null,
cloudCoverPhrase: "Clear",
dayOfWeek: "Wednesday",
dayOrNight: "D",
expirationTimeUtc: 1545249077,
iconCode: 32,
iconCodeExtend: 3200,
obsQualifierCode: null,
obsQualifierSeverity: null,
precip1Hour: 0,
precip6Hour: 0,
precip24Hour: 0,
pressureAltimeter: 1018.29,
pressureChange: -2.71,
pressureMeanSeaLevel: 1018.5,
pressureTendencyCode: 2,
pressureTendencyTrend: "Falling",
relativeHumidity: 55,
snow1Hour: 0,
snow6Hour: 0,
snow24Hour: 0,
sunriseTimeLocal: "2018-12-19T07:28:58-0500",
sunriseTimeUtc: 1545222538,
sunsetTimeLocal: "2018-12-19T17:10:52-0500",
sunsetTimeUtc: 1545257452,
```

```
temperature: 10,  
temperatureChange24Hour: -2,  
temperatureDewPoint: 2,  
temperatureFeelsLike: 9,  
temperatureHeatIndex: 10,  
temperatureMax24Hour: 12,  
temperatureMaxSince7Am: 10,  
temperatureMin24Hour: -3,  
temperatureWindChill: 9,  
uvDescription: "Low",  
uvIndex: 1,  
validTimeLocal: "2018-12-19T14:41:17-0500",  
validTimeUtc: 1545248477,  
visibility: 16.09,  
windDirection: 180,  
windDirectionCardinal: "S",  
windGust: null,  
windSpeed: 6,  
wxPhraseLong: "Sunny",  
wxPhraseMedium: "Sunny",  
wxPhraseShort: "Sunny"  
}
```

284.1.28 getTwcForecast()

Note: If you are considering the development of an application that makes extensive use of weather data, you should consider gaining direct access to APIs from a weather data provider.

Get the daily weather forecast at the specified location.

Signature: `def getTwcForecast(String locationString=null)`

Parameters: `String locationString` - Optional. Must be a 5 digit US zip code or a latitude, longitude string (e.g., "38.25,-76.45"). If not specified, the method will use the latitude and longitude of the Location as set in the SmartThings mobile app.

Example Response:

```
{  
  dayOfWeek: [  
    "Wednesday",  
    "Thursday",  
    "Friday",  
    "Saturday"  
  ],  
  expirationTimeUtc: [  
    1545251268,  
    1545251268,  
    1545251268,  
    1545251268  
  ],  
  moonPhase: [  
    "Waxing Gibbous",  
    "Waxing Gibbous"  
  ],  
}
```



```

    "Waxing Gibbous",
    "Full Moon"
  ],
  moonPhaseCode: [
    "WXG",
    "WXG",
    "WXG",
    "F"
  ],
  moonPhaseDay: [
    11,
    12,
    13,
    15
  ],
  moonriseTimeLocal: [
    "2018-12-19T15:04:06-0500",
    "2018-12-20T15:44:43-0500",
    "2018-12-21T16:32:25-0500",
    "2018-12-22T17:26:58-0500"
  ],
  moonriseTimeUtc: [
    1545249846,
    1545338683,
    1545427945,
    1545517618
  ],
  moonsetTimeLocal: [
    "2018-12-19T03:50:48-0500",
    "2018-12-20T04:56:24-0500",
    "2018-12-21T06:03:51-0500",
    "2018-12-22T07:11:16-0500"
  ],
  moonsetTimeUtc: [
    1545209448,
    1545299784,
    1545390231,
    1545480676
  ],
  narrative: [
    "A few clouds. Highs in the low 50s and lows in the upper 30s.",
    "Cloudy, periods of rain. Highs in the upper 40s with temperatures nearly steady overnight.",
    "Cloudy with rain. Highs in the mid 50s and lows in the upper 30s.",
    "Mostly sunny. Highs in the upper 40s and lows in the low 30s."
  ],
  qpf: [
    0,
    1.44,
    0.49,
    0
  ],
  qpfSnow: [
    0,
    0,
    0,
    0
  ],
  sunriseTimeLocal: [

```

```
    "2018-12-19T07:28:58-0500",
    "2018-12-20T07:29:31-0500"
    "2018-12-21T07:30:02-0500"
    "2018-12-22T07:30:32-0500"
  ],
  sunriseTimeUtc:[
    1545222538,
    1545308971,
    1545395402,
    1545481832
  ],
  sunsetTimeLocal:[
    "2018-12-19T17:10:52-0500",
    "2018-12-20T17:11:19-0500"
    "2018-12-21T17:11:47-0500",
    "2018-12-22T17:12:18-0500"
  ],
  sunsetTimeUtc:[
    1545257452,
    1545343879,
    1545430307,
    1545516738
  ],
  temperatureMax:[
    51,
    49,
    54,
    49
  ],
  temperatureMin:[
    38,
    47,
    37,
    31
  ],
  validTimeLocal:[
    "2018-12-19T07:00:00-0500",
    "2018-12-20T07:00:00-0500",
    "2018-12-21T07:00:00-0500",
    "2018-12-22T07:00:00-0500"
  ],
  validTimeUtc:[
    1545220800,
    1545307200,
    1545393600,
    1545480000
  ],
  daypart:[
    (
      cloudCover:[
        16,
        79,
        100,
        100,
        99,
        85,
        32,
        14
```

```
    },
    dayOrNight: [
        "D",
        "N",
        "D",
        "N",
        "D",
        "N",
        "D",
        "N"
    ],
    daypartName: [
        "Today",
        "Tonight",
        "Tomorrow",
        "Tomorrow night",
        "Friday",
        "Friday night",
        "Saturday",
        "Saturday night"
    ],
    iconCode: [
        34,
        27,
        12,
        12,
        12,
        26,
        34,
        33
    ],
    iconCodeExtend: [
        3400,
        2700,
        1200,
        1200,
        1200,
        2600,
        3400,
        3300
    ],
    narrative: [
        "Lots of sunshine. High 51F. Winds light and variable.",
        "Partly cloudy early followed by cloudy skies overnight. Low 38F. Winds light and variable.",
        "Rain likely. High 49F. Winds NE at 5 to 10 mph. Chance of rain 100%. Rainfall near a",
        "Rain likely. Low 47F. Winds light and variable. Chance of rain 90%. Rainfall near a",
        "Periods of rain. Thunder possible. High 54F. Winds SSW at 5 to 10 mph. Chance of rain",
        "Cloudy. Low 37F. Winds WNW at 5 to 10 mph.",
        "A few clouds early, otherwise mostly sunny. High 49F. Winds WNW at 5 to 10 mph.",
        "Clear to partly cloudy. Low 31F. Winds light and variable."
    ],
    precipChance: [
        0,
        20,
        100,
        90,
        100,
        20,
```

```
    0,  
    0  
  ],  
  precipType: [  
    "rain",  
    "precip",  
    "rain",  
    "rain",  
    "rain",  
    "precip",  
    "rain",  
    "precip"  
  ],  
  qpf: [  
    0,  
    0,  
    0.93,  
    0.51,  
    0.48,  
    0,  
    0,  
    0  
  ],  
  qpfSnow: [  
    0,  
    0,  
    0,  
    0,  
    0,  
    0,  
    0,  
    0,  
    0  
  ],  
  qualifierCode: [  
    null,  
    null,  
    null,  
    null,  
    "Q8003",  
    null,  
    null,  
    null  
  ],  
  qualifierPhrase: [  
    null,  
    null,  
    null,  
    null,  
    "Thunder possible.",  
    null,  
    null,  
    null  
  ],  
  relativeHumidity: [  
    63,  
    85,  
    93,  
    96,  
  ]  
}
```

```
    92,  
    76,  
    55,  
    72  
  ],  
  snowRange: [  
    "",  
    "",  
    "",  
    "",  
    "",  
    "",  
    "",  
    ""  
  ],  
  temperature: [  
    51,  
    38,  
    49,  
    47,  
    54,  
    37,  
    49,  
    31  
  ],  
  temperatureHeatIndex: [  
    50,  
    43,  
    48,  
    50,  
    54,  
    46,  
    48,  
    39  
  ],  
  temperatureWindChill: [  
    44,  
    39,  
    41,  
    46,  
    43,  
    34,  
    33,  
    32  
  ],  
  thunderCategory: [  
    "No thunder",  
    "No thunder",  
    "No thunder",  
    "No thunder",  
    "Thunder possible",  
    "No thunder",  
    "No thunder",  
    "No thunder"  
  ],  
  thunderIndex: [  
    0,  
    0,  
    0,  
    0,  
    0,  
    0,  
    0,  
    0
```



```
    locale1:null,  
    locale2:"Boonville",  
    locale3:null,  
    locale4:null  
  },  
  neighborhood:null,  
  adminDistrict:"North Carolina",  
  adminDistrictCode:"NC",  
  postalCode:"27011",  
  postalKey:"27011:US",  
  country:"United States",  
  countryCode:"US",  
  ianaTimeZone:"America/New_York",  
  displayName:"Boonville",  
  dstEnd:"2019-11-03T01:00:00-0500",  
  dstStart:"2019-03-10T03:00:00-0400",  
  dmaCd:"518",  
  placeId:"5a75bd28971b1f181dde5085446a99e4abf9adbf1754365bb5dc9ac53d6779a4",  
  disputedArea:false,  
  countyId:"NCC197",  
  locId:null,  
  pwsId:null,  
  type:"postal",  
  zoneId:"NCZ020"  
}
```

284.1.30 getTwcAlerts()

Note: If you are considering the development of an application that makes extensive use of weather data, you should consider gaining direct access to APIs from a weather data provider.

Get the current severe weather alerts at the specified location.

Signature: `def getTwcAlerts(String geoLocation=null)`

Parameters: `String geoLocation` - Optional. A latitude and longitude string (e.g., "38.25,-76.45"). Zip codes are not supported by `getTwcAlerts()`.

Example Response:

```
{  
  metadata:{  
    next:null  
  },  
  alerts:[  
    {  
      detailKey:"c991e7f1-7519-3501-9481-dce00c81bb9e",  
      messageTypeCode:2,  
      messageType:"Update",  
      productIdentifier:"FLS",  
      phenomena:"FL",  
      significance:"W",  
      eventTrackingNumber:"0087",  
    }  
  ]  
}
```



```

officeCode:"KLCH",
officeName:"Lake Charles",
officeAdminDistrict:"Louisiana",
officeAdminDistrictCode:"LA",
officeCountryCode:"US",
eventDescription:"River Flood Warning",
severityCode:3,
severity:"Moderate",
categories:[
  {
    category:"Met",
    categoryCode:2
  }
],
responseTypes:[
  {
    responseType:"Avoid",
    responseTypeCode:5
  }
],
urgency:"Unknown",
urgencyCode:5,
certainty:"Unknown",
certaintyCode:5,
effectiveTimeLocal:null,
effectiveTimeLocalTimeZone:null,
expireTimeLocal:"2018-12-20T00:50:00-06:00",
expireTimeLocalTimeZone:"CST",
expireTimeUTC:1545288600,
onsetTimeLocal:null,
onsetTimeLocalTimeZone:null,
flood:[
  {
    floodLocationId:"DWYT2",
    floodLocationName:"Sabine River near Deweyville",
    floodSeverityCode:"1",
    floodSeverity:"Minor",
    floodImmediateCauseCode:"ER",
    floodImmediateCause:"Excessive Rainfall",
    floodRecordStatusCode:"NO",
    floodRecordStatus:"A record flood is not expected",
    floodStartTimeLocal:"2018-11-04T05:07:00-06:00",
    floodStartTimeLocalTimeZone:"CST",
    floodCrestTimeLocal:"2018-12-13T09:00:00-06:00",
    floodCrestTimeLocalTimeZone:"CST",
    floodEndTimeLocal:null,
    floodEndTimeLocalTimeZone:null
  }
],
areaTypeCode:"C",
latitude:30.23,
longitude:-93.33,
areaId:"LAC019",
areaName:"Calcasieu Parish",
ianaTimeZone:"America/Chicago",
adminDistrictCode:"LA",
adminDistrict:"Louisiana",
countryCode:"US",
countryName:"UNITED STATES OF AMERICA",
headlineText:"River Flood Warning is in effect",

```

```
source:"National Weather Service",
disclaimer:null,
issueTimeLocal:"2018-12-19T10:51:00-06:00",
issueTimeLocalTimeZone:"CST",
identifier:"e36df9092f95582ad3b5021bbc480481",
processTimeUTC:1545238316
```

284.1.31 getTwcAlertDetail()

Note: If you are considering the development of an application that makes extensive use of weather data, you should consider gaining direct access to APIs from a weather data provider.

Get detailed description and text of the specified weather alert.

Signature: def getTwcAlertDetail(String alertId)

Parameters: String alertId - The *alertId* from the response from *getTwcAlerts()*.

Example Response:

```
alertDetail: {
  detailKey: "c991e7f1-7519-3501-9481-dce00c81bb9e",
  messageTypeCode: 2,
  messageType: "Update",
  productIdentifier: "FLS",
  phenomena: "FL",
  significance: "W",
  eventTrackingNumber: "0087",
  officeCode: "KLCH",
  officeName: "Lake Charles",
  officeAdminDistrict: "Louisiana",
  officeAdminDistrictCode: "LA",
  officeCountryCode: "US",
  eventDescription: "River Flood Warning",
  severityCode: 3,
  severity: "Moderate",
  categories: [
    {
      category: "Met",
      categoryCode: 2
    }
  ],
  responseTypes: [
    {
      responseType: "Avoid",
      responseTypeCode: 5
    }
  ],
  urgency: "Unknown",
  urgencyCode: 5,
```

```

certainty:"Unknown",
certaintyCode:5,
effectiveTimeLocal:null,
effectiveTimeLocalTimeZone:null,
expireTimeLocal:"2018-12-20T00:50:00-06:00",
expireTimeLocalTimeZone:"CST",
expireTimeUTC:1545288600,
onsetTimeLocal:null,
onsetTimeLocalTimeZone:null,
flood: {
  floodLocationId:"DWYT2",
  floodLocationName:"Sabine River near Deweyville",
  floodSeverityCode:"1",
  floodSeverity:"Minor",
  floodImmediateCauseCode:"ER",
  floodImmediateCause:"Excessive Rainfall",
  floodRecordStatusCode:"NO",
  floodRecordStatus:"A record flood is not expected",
  floodStartTimeLocal:"2018-11-04T05:07:00-06:00",
  floodStartTimeLocalTimeZone:"CST",
  floodCrestTimeLocal:"2018-12-13T09:00:00-06:00",
  floodCrestTimeLocalTimeZone:"CST",
  floodEndTimeLocal:null,
  floodEndTimeLocalTimeZone:null
},
areaTypeCode:"C",
latitude:30.23,
longitude:-93.33,
areaId:"LAC019",
areaName:"Calcasieu Parish",
ianaTimeZone:"America/Chicago",
adminDistrictCode:"LA",
adminDistrict:"Louisiana",
countryCode:"US",
countryName:"UNITED STATES OF AMERICA",
headlineText:"River Flood Warning is in effect",
source:"National Weather Service",
disclaimer:null,
issueTimeLocal:"2018-12-19T10:51:00-06:00",
issueTimeLocalTimeZone:"CST",
identifier:"e36df9092f95582ad3b5021bbc480481",
processTimeUTC:1545238316,
texts: [
  {
    languageCode:"en-US",
    description:"The Flood Warning continues for The Sabine River Near Deweyville. * until",
    instruction:null,
    overview:"...The Flood Warning continues for the following rivers in Texas... Neches
  }
],
polygon: [
  {
    lat:30.57,
    lon:-93.63
  },
  {
    lat:30.11,
    lon:-93.64
  }
]

```

```
    lat:30.11,  
    lon:-93.78  
  },  
  {  
    lat:30.31,  
    lon:-93.81  
  },  
  {  
    lat:30.62,  
    lon:-93.78  
  },  
  {  
    lat:30.57,  
    lon:-93.63  
  }  
],  
synopsis:null
```

284.1.32 `getWeatherFeature()` - Deprecated

Warning: Effective January 1, 2019, this API will be removed. See `getTwcConditions()` (page 907), `getTwcForecast()` (page 908), `getTwcAlerts()` (page 916), `getTwcLocation()` (page 915), or `getTwcAlertDetail()` (page 918) for alternative weather APIs.

Calls the Weather Underground API to to return weather forecasts and related data.

Signature: `Map getWeatherFeature(String featureName [, String location])`

Note: `getWeatherFeature` simply delegates to the Weather Underground API, using the specified `featureName` and `location` (if specified). For full descriptions on the available features and return information, please consult the [Weather Underground API docs](#).

Parameters: `String featureName` The weather feature to get. This corresponds to the available “Data Features” in the Weather Underground API.

`String location` (*optional*) The location to get the weather information for (ZIP code). If not specified, the Location of the user’s Hub will be used.

Returns: `Map` - a Map containing the weather information requested. The data returned will vary depending on the feature requested. See the Weather Underground API documentation for more information.

284.1.33 `httpDelete()`

Executes an HTTP DELETE request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

Signature: `void httpDelete(String uri, Closure closure)`

`void httpDelete(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP DELETE call to.

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Forced response content type and request Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

Returns: `void`

284.1.34 httpError()

Throws a `SmartAppException` with the specified status code and message.

This should be used to send an HTTP error to any calling client.

Signature: `def httpError(Integer status, message)`

Parameters: `Integer status` - The HTTP error code to send. `message` - the error message.

Example:

```
def someMethod() {
    httpError(400, "something went wrong")
}
```

284.1.35 httpGet()

Executes an HTTP GET request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpGet(String uri, Closure closure)`

`void httpGet(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP GET call to

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Forced response content type and request Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure - closure - The closure that will be called with the response of the request.

Example:

```
def params = [
    uri: "http://httpbin.org",
    path: "/get"
]

try {
    httpGet(params) { resp ->
        resp.headers.each {
            log.debug "${it.name} : ${it.value}"
        }
        log.debug "response contentType: ${resp.contentType}"
        log.debug "response data: ${resp.data}"
    }
} catch (e) {
    log.error "something went wrong: $e"
}
```

284.1.36 httpHead()

Executes an HTTP HEAD request and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

Signature: void httpHead(String uri, Closure closure)

void httpHead(Map params, Closure closure)

Parameters: String uri - The URI to make the HTTP HEAD call to

Map params - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Forced response content type and request Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure closure - The closure that will be called with the response of the request.

284.1.37 httpPost()

Executes an HTTP POST request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPost(String uri, String body, Closure closure)`
`void httpPost(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP POST call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Forced response content type and request Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

Example:

```
try {
    httpPost("http://mysite.com/api/call", "id=XXX&value=YYY") { resp ->
        log.debug "response data: ${resp.data}"
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.debug "something went wrong: $e"
}
```

284.1.38 httpPostJson()

Executes an HTTP POST request with a JSON-encoded body and content type, and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPostJson(String uri, String body, Closure closure)`
`void httpPostJson(String uri, Map body, Closure closure)`
`void httpPostJson(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP POST call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Forced response content type and request Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure `closure` - The closure that will be called with the response of the request.

Example:

```
def params = [
  uri: "http://postcatcher.in/catchers/<yourUniquePath>",
  body: [
    param1: [subparam1: "subparam 1 value",
             subparam2: "subparam2 value"],
    param2: "param2 value"
  ]
]

try {
  httpPostJson(params) { resp ->
    resp.headers.each {
      log.debug "${it.name} : ${it.value}"
    }
    log.debug "response contentType: ${resp.contentType}"
  }
} catch (e) {
  log.debug "something went wrong: $e"
}
```

284.1.39 httpPut()

Executes an HTTP PUT request and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPut(String uri, String body, Closure closure)`

`void httpPut(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP PUT call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Forced response content type and request Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure *closure* - The closure that will be called with the response of the request.

Example:

```
try {
    httpPut("http://mysite.com/api/call", "id=XXX&value=YYY") { resp ->
        log.debug "response data: ${resp.data}"
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.error "something went wrong: $e"
}
```

284.1.40 httpPutJson()

Executes an HTTP PUT request with a JSON-encoded body and content type, and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPutJson(String uri, String body, Closure closure)`

`void httpPutJson(String uri, Map body, Closure closure)`

`void httpPutJson(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP PUT call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Forced response content type and request Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure *closure* - The closure that will be called with the response of the request.

284.1.41 nextOccurrence()

Returns a Date when the time specified in the input occurs next.

Signature: Date nextOccurrence(String timeString)

Parameters: String timeString - An ISO-8601 date string as returned from time input preferences of the SmartApp.

Note: Note that if the input timeString does not contain time zone, this method will throw an IllegalArgumentException.

Returns: Date - The Date when the time specified in the timeString occurs next. If the specified time has already occurred, then returns the next day Date object when the specified time occurs next. If the specified time has not yet occurred, then returns today's Date object when the specified time will occur.

Example:

```
preferences {
    section() {
        input "Time1", "time", title: "Time1"
        input "Time2", "time", title: "Time2"
    }
}
...

// Current time is 16:25 October 24, 2016, Time1 input is 16:23 and Time2 input is 16:34
log.debug "nextOccurrence(Time1) value is: ${nextOccurrence(Time1)}"
log.debug "nextOccurrence(Time2) value is: ${nextOccurrence(Time2)}"
// The above log statements will print the following:
nextOccurrence(Time1) value is: Tue Oct 25 23:23:00 UTC 2016
nextOccurrence(Time2) value is: Mon Oct 24 23:34:00 UTC 2016
```

284.1.42 now()

Gets the current Unix time in milliseconds.

Signature: Long now()

Returns: Long - the current Unix time.

284.1.43 parseJson()

Parses the specified string into a JSON data structure.

Signature: Map parseJson(stringToParse)

Parameters: String stringToParse - The string to parse into JSON

Returns: Map - a map that represents the passed-in string in JSON format.

284.1.44 parseXml()

Parses the specified string into an XML data structure.

Signature: `GPathResult parseXml(stringToParse)`

Parameters: `String stringToParse` - The string to parse into XML

Returns: `GPathResult` - A `GPathResult` instance that represents the passed-in string in XML format.

284.1.45 parseLanMessage()

Parses a Base64-encoded LAN message received from the Hub into a map with header and body elements, as well as parsing the body into an XML document.

Signature: `Map parseLanMessage(stringToParse)`

Parameters: `String stringToParse` - The string to parse

Returns: `Map` - a map with the following structure:

key	type	description
header	<code>String</code>	the headers of the request as a single string
headers	<code>Map</code>	a Map of string/name value pairs for each header
body	<code>String</code>	the request body as a string

284.1.46 parseSoapMessage()

Parses a Base64-encoded LAN message received from the Hub into a map with header and body elements, as well as parsing the body into an XML document. This method is commonly used to parse [UPNP SOAP](#) messages.

Signature: `Map parseLanMessage(stringToParse)`

Parameters: `String stringToParse` - The string to parse

Returns: `Map` - A map with the following structure:

key	type	description
header	<code>String</code>	the headers of the request as a single string
headers	<code>Map</code>	a Map of string/name value pairs for each header
body	<code>String</code>	the request body as a string
xml	<code>GPathResult</code>	the request body as a <code>GPathResult</code> object
xmlError	<code>String</code>	error message from parsing the body, if any

284.1.47 render()

Returns a HTTP response to the calling client with the options specified.

Signature: `def render(Map options)`

Parameters: `Map` options - the options for what is returned to the client:

option	description
<code>contentType</code>	The value of the “Content-Type” request header. “application/json” if not specified.
<code>status</code>	The HTTP status of the response. 200 if not specified.
<code>data</code>	Required. The data for this response.

Example:

```
def someMethod() {
  def html = """
    <!DOCTYPE HTML>
    <html>
      <head><title>Some Title</title></head>
      <body><p>Some Text</p></body>
    </html>
  """

  render contentType: "text/html", data: html
}
```

284.1.48 revokeAccessToken()

Revokes the access token created with `createAccessToken()` (page 902) for this installed SmartApp.

Signature: `def revokeAccessToken()`

Example:

```
// Check to see if SmartApp has its own access token and create one if not.
if(!state.accessToken) {
  // the createAccessToken() method will store the access token in state.accessToken
  createAccessToken()
}

// Use token to allow third-party to communicate with SmartApp during setup

// Revoke the token once the third-party no longer needs it (after setup)
revokeAccessToken()
```

See also:

- *Building the Service Manager* (page 546)
- `createAccessToken()` (page 902)

284.1.49 runIn()

Executes a specified handlerMethod after delaySeconds have elapsed.

Signature: `void runIn(delayInSeconds, handlerMethod [, options])`

Tip: It's important to note that we will attempt to run this method at this time, but cannot guarantee exact precision. We typically expect per-minute level granularity, so if using with values less than sixty seconds, your mileage will vary.

Parameters: `delayInSeconds` - The number of seconds to execute the `handlerMethod` after.

`handlerMethod` - The method to call after `delayInSeconds` has passed. Can be a string or a reference to the method. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
overwrite	true or false	Specify <code>[overwrite: false]</code> to not overwrite any existing pending schedule handler for the given method (the default behavior is to overwrite the pending schedule). Specifying <code>[overwrite: false]</code> can lead to multiple different schedules for the same handler method, so be sure your handler method can handle this.
data	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
runIn(300, myHandlerMethod)
runIn(400, "myOtherHandlerMethod", [data: [flag: true]])

def myHandlerMethod() {
    log.debug "handler method called"
}

def myOtherHandlerMethod(data) {
    log.debug "other handler method called with flag: $data.flag"
}
```

284.1.50 runEvery1Minute()

Creates a recurring schedule that executes the specified `handlerMethod` every minute. Using this method will pick a random start time in the next minute, and run every minute after that.

Signature: void `runEvery1Minute(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the 1 minute period.

Parameters: `handlerMethod` - The method to call every minute. Can be the name of the method as a string, or a reference to the method. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
data	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
runEvery1Minute(handlerMethod1)
runEvery1Minute(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.1.51 runEvery5Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every five minutes. Using this method will pick a random start time in the next five minutes, and run every five minutes after that.

Signature: void runEvery5Minutes(handlerMethod[, options])

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the 5 minute period.

Parameters: `handlerMethod` - The method to call every five minutes. Can be the name of the method as a string, or a reference to the method. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
data	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
runEvery5Minutes(handlerMethod1)
runEvery5Minutes(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.1.52 runEvery10Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every ten minutes. Using this method will pick a random start time in the next ten minutes, and run every ten minutes after that.

Signature: `void runEvery10Minutes(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the ten minute period.

Parameters: `handlerMethod` - The method to call every ten minutes. Can be the name of the method as a string, or a reference to the method. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: `void`

Example:

```
runEvery10Minutes(handlerMethod1)
runEvery10Minutes(handlerMethod2, [data: [key: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.1.53 runEvery15Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every fifteen minutes. Using this method will pick a random start time in the next five minutes, and run every five minutes after that.

Signature: `void runEvery15Minutes(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the fifteen minute period.

Parameters: `handlerMethod` - The method to call every fifteen minutes. Can be the name of the method as a string, or a reference to the method. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
runEvery15Minutes(handlerMethod1)
runEvery15Minutes(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.1.54 runEvery30Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every thirty minutes. Using this method will pick a random start time in the next thirty minutes, and run every thirty minutes after that.

Signature: void runEvery30Minutes(handlerMethod[, options])

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the thirty minute period.

Parameters: `handlerMethod` - The method to call every thirty minutes. Can be the name of the method as a string, or a reference to the method. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
data	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
runEvery30Minutes(handlerMethod1)
runEvery30Minutes(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.1.55 runEvery1Hour()

Creates a recurring schedule that executes the specified `handlerMethod` every hour. Using this method will pick a random start time in the next hour, and run every hour after that.

Signature: `void runEvery1Hour(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the one hour period.

Parameters: `handlerMethod`- The method to call every hour. Can be the name of the method as a string, or a reference to the method. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: `void`

Example:

```
runEvery1Hour(handlerMethod1)
runEvery1Hour(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.1.56 runEvery3Hours()

Creates a recurring schedule that executes the specified `handlerMethod` every three hours. Using this method will pick a random start time in the next hour, and run every three hours after that.

Signature: `void runEvery3Hours(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the three hour period.

Parameters: `handlerMethod` - The method to call every three hours. Can be the name of the method as a string, or a reference to the method. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
runEvery3Hours(handlerMethod1)
runEvery3Hours(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.1.57 runOnce()

Executes the handlerMethod once at the specified date and time.

Signature: void runOnce(dateTime, handlerMethod [, options])

Parameters: dateTime - When to execute the handlerMethod. Can be either a [Date](#) object or an ISO-8601 date string. For example, `new Date() + 1` would run at the current time tomorrow, and `"2017-07-04T12:00:00.000Z"` would run at noon GMT on July 4th, 2017.

handlerMethod - The method to execute at the specified dateTime. This can be a reference to the method, or the method name as a string. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

options (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
overwrite	true or false	Specify <code>[overwrite: false]</code> to not overwrite any existing pending schedule handler for the given method (the default behavior is to overwrite the pending schedule). Specifying <code>[overwrite: false]</code> can lead to multiple different schedules for the same handler method, so be sure your handler method can handle this.
data	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
// execute handler at 4 PM CST on October 21, 2015 (e.g., Back to the Future 2 Day!)
runOnce("2015-10-21T16:00:00.000-0600", handler)

def handler() {
    ...
}
```

284.1.58 schedule()

Creates a scheduled job that calls the `handlerMethod` once per day at the time specified, or according to a cron schedule.

Signature: `void schedule(dateTime, handlerMethod [, options])`
`void schedule(cronExpression, handlerMethod [, options])`

Parameters:

`dateTime` - A `Date` object, an ISO-8601 formatted date time string.

`String cronExpression` - A cron expression that specifies the schedule to execute on.

`handlerMethod` - The method to call. This can be a reference to the method itself, or the method name as a string. Make sure that the method being referenced is not scoped to private and/or the method name being used does not include parens (e.g. `handlerMethod()`)

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: `void`

Tip: Since calling `schedule()` with a `dateTime` argument creates a recurring scheduled job to execute *every day* at the specified time, the *date information is ignored. Only the time portion of the argument is used.*

Tip: Full documentation for the cron expression format can be found in the [Quartz Cron Trigger Tutorial](#)

Example:

```

preferences {
    section() {
        input "timeToRun", "time"
    }
}

...
// call handlerMethod1 at time specified by user input
schedule(timeToRun, handlerMethod1)

// call handlerMethod2 every day at 3:36 PM CST
schedule("2015-01-09T15:36:00.000-0600", handlerMethod2)

// execute handlerMethod3 every hour on the half hour (using a randomly chosen seconds field)
schedule("12 30 * * * ?", handlerMethod3)
...

def handlerMethod1() {...}
def handlerMethod2() {...}
def handlerMethod3() {...}

```

284.1.59 sendEvent()

Creates and sends an Event constructed from the specified properties. If a device is specified, then a DEVICE Event will be created, otherwise an APP Event will be created.

Note: SmartApps typically *respond to Events*, not create them. In more rare cases, certain SmartApps or Service Manager SmartApps may have reason to send Events themselves. `sendEvent` can be used for those cases.

Signature: `void sendEvent(Map properties)`

`void sendEvent(Device device, Map properties)`

Parameters: `Map` properties - The properties of the Event to create and send.

Here are the available properties:

Property	Description
name (required)	<code>String</code> - The name of the Event. Typically corresponds to an attribute name of a capability.
value (required)	The value of the Event. The value is stored as a string, but you can pass numbers or other objects.
descriptionText	<code>String</code> - The description of this Event. This appears in the mobile application activity for the device. If not specified, this will be created using the Event name and value.
displayed	Pass <code>true</code> to display this Event in the mobile application activity feed, <code>false</code> to not display. Defaults to <code>true</code> .
linkText	<code>String</code> - Name of the Event to show in the mobile application activity feed.
isStateChange	<code>true</code> if this Event caused a device attribute to change state. Typically not used, since it will be set automatically.
unit	<code>String</code> - a unit string, if desired. This will be used to create the <code>descriptionText</code> if it (the <code>descriptionText</code> option) is not specified.
<i>Device</i> (page 1003)	<code>device</code> - The device for which this Event is created for.
data	A map of additional information to store with the Event

Tip: Not all Event properties need to be specified. ID properties like `deviceId` and `locationId` are automatically set, as are properties like `isStateChange`, `displayed`, and `linkText`.

Returns: `void`

Example:

```
// create and send an event with name "temperature" and value 72
sendEvent(name: "temperature", value: 72, unit: "F")

// create and send event with additional data
sendEvent(name: "myevent", value: "myvalue", data: [moreInfo: "more information", evenMoreInfo: 42])
```

284.1.60 sendHubCommand()

Sends a command to the Hub, with the details of the command encapsulated within a `HubAction` object.

Signature: `void sendHubCommand(HubAction action)`

`void sendHubCommand(List<HubAction> actions, delay)`

Parameters: `HubAction action` - A `HubAction` object

`List<HubAction> actions` - A list of `HubAction` objects

`delay` - An integer number representing milliseconds. This is the delay between commands when a list of `HubAction` objects are sent using `List<HubAction> actions` parameter. The default value of `delay` is 1000.

Returns: `void`

Example: During the discovery phase of a LAN-connected device the following discovery command can be sent to the Hub.

```
// Send a single HubAction command to the Hub
void ssdpDiscover() {
    sendHubCommand(new physicalgraph.device.HubAction("lan discovery urn:schemas-upnp-org:device:ZonePl
})
// Send a List of HubAction commands to the Hub with a delay of 3 seconds between each HubAction com
void sendMultiDevice() {
    List actions = []
    actions.add(new physicalgraph.device.HubAction("lan discovery urn:schemas-upnp-org:device:ZonePl
    actions.add(new physicalgraph.device.HubAction("lan discovery urn:schemas-upnp-org:device:MediaR
    actions.add(new physicalgraph.device.HubAction("lan discovery urn:samsung.com:device:RemoteContro
    sendHubCommand(actions, 3000)
}
```

284.1.61 sendLocationEvent()

Sends a `LOCATION` Event constructed from the specified properties. See the [Event](#) (page 1017) reference for a list of available properties. Other SmartApps can receive Location Events by subscribing to the Location. Examples of existing Location Events include sunrise and sunset.

Signature: `void sendLocationEvent(Map properties)`

Parameters: `Map properties` - The properties from which to create and send the Event.

Here are the available properties:

Property	Description
<code>name</code> (required)	<code>String</code> - The name of the Event. Typically corresponds to an attribute name of a capability.
<code>value</code> (required)	The value of the Event. The value is stored as a string, but you can pass numbers or other objects.
<code>descriptionText</code>	<code>String</code> - The description of this Event. This appears in the mobile application activity for the device. If not specified, this will be created using the Event name and value.
<code>displayed</code>	Pass <code>true</code> to display this Event in the mobile application activity feed, <code>false</code> to not display. Defaults to <code>true</code> .
<code>linkText</code>	<code>String</code> - Name of the Event to show in the mobile application activity feed.
<code>isStateChange</code>	<code>true</code> if this Event caused a device attribute to change state. Typically not used, since it will be set automatically.
<code>unit</code>	<code>String</code> - a unit string, if desired. This will be used to create the <code>descriptionText</code> if it (the <code>descriptionText</code> option) is not specified.
<code>data</code>	A map of additional information to store with the Event

Returns: void

284.1.62 sendNotification()

Sends the specified message and displays it in the *Hello, Home* portion of the mobile application.

Signature: void sendNotification(String message [, Map options])

Parameters: String message - The message to send to *Hello, Home*

Map options (*optional*) - Options for the message. The following options are available:

option	description
method	String - One of "phone", "push", or "both". Defaults to "both".
event	false to suppress displaying in <i>Hello, Home</i> . Defaults to true.
phone	String - The phone number to send the SMS message to. Required when the method is "phone". If not specified and method is "both", then no SMS message will be sent.

Returns: void

Example:

```
sendNotification("test notification - no params")
sendNotification("test notification - push", [method: "push"])
sendNotification("test notification - sms", [method: "phone", phone: "1234567890"])
sendNotification("test notification - both", [method: "both", phone: "1234567890"])
sendNotification("test notification - no event", [event: false])
```

284.1.63 sendNotificationEvent()

Displays a message in *Hello, Home*, but does not send a push notification or SMS message.

Signature: void sendNotificationEvent(String message)

Parameters: String message - The message to send to *Hello, Home*

Returns: void

Example:

```
sendNotificationEvent("some message")
```

284.1.64 sendNotificationToContacts()

Sends the specified message to the specified contacts.

Signature: void sendNotificationToContacts(String message, String contact, Map options={:})

void sendNotificationToContacts(String message, Collection contacts, Map options={:})

Parameters: `String` `message` - the message to send

`String` `contact` - the contact to send the notification to. Typically set through the `contact` input type.

`Collection` `contacts` - the collection of contacts to send the notification to. Typically set through the `contact` input type.

`Map` `options` (*optional*) - a map of additional parameters. The valid parameter is `[event: boolean]` to specify if the message should be displayed in the Notifications feed. Defaults to `true` (message will be displayed in the Notifications feed).

Returns: `void`

Example:

```
preferences {
    section("Send Notifications?") {
        input("recipients", "contact", title: "Send notifications to") {
            input "phone", "phone", title: "Warn with text message (optional)",
                description: "Phone Number", required: false
        }
    }
}

...
if (location.contactBookEnabled) {
    sendNotificationToContacts("Your house talks!", recipients)
}

...
```

Tip: It's a good idea to assume that a user *may not* have any contacts configured. That's why you see the nested "phone" input in the preferences (user will only see that if they don't have contacts), and why we check `location.contactBookEnabled`.

284.1.65 sendPush()

Sends the specified message as a push notification to users mobile devices and displays it in *Hello, Home*.

Signature: `void sendPush(String message)`

Parameters: `String` `message` - The message to send

Returns: `void`

Example:

```
sendPush("some message")
```

284.1.66 sendPushMessage()

Sends the specified message as a push notification to users mobile devices but does not display it in *Hello, Home*.

Signature: `void sendPushMessage(String message)`

Parameters: `String` `message` - The message to send

Returns: `void`

Example:

```
sendPushMessage ("some message")
```

284.1.67 sendSms()

Sends the message as an SMS message to the specified phone number and displays it in Hello, Home. The message can be no longer than 140 characters.

Signature: `void sendSms(String phoneNumber, String message)`

Parameters: `String phoneNumber` - the phone number to send the SMS message to.

`String message` - the message to send. Can be no longer than 140 characters.

Returns: `void`

Example:

```
sendSms ("somePhoneNumber", "some message")
```

284.1.68 sendSmsMessage()

Sends the message as an SMS message to the specified phone number but does not display it in Hello, Home. The message can be no longer than 140 characters.

Signature: `void sendSmsMessage(String phoneNumber, String message)`

Parameters: `String phoneNumber` - the phone number to send the SMS message to.

`String message` - the message to send. Can be no longer than 140 characters.

Returns: `void`

Example:

```
sendSms ("somePhoneNumber", "some message")
```

284.1.69 setLocationMode()

Set the Mode for this Location.

Signature: `void setLocationMode(String mode) void setLocationMode(Mode mode)`

Returns: `void`

Warning: `setMode()` will raise an error if the specified Mode does not exist for the Location. You should verify the Mode exists as in the example below.

See Also: [location.setMode\(\)](#) (page 1038)

284.1.70 settings

A map of name/value pairs containing all of the installed SmartApp's preferences.

Signature: Map settings

Returns: Map - a map containing all of the installed SmartApp's preferences.

Example:

```
preferences {
    section() {
        input "myswitch", "capability.switch"
        input "mytext", "text"
        input "mytime", "time"
    }
}

...

log.debug "settings.mytext: ${settings.mytext}"
log.debug "settings.mytime: ${settings.mytime}"

// if the input is a device/capability, you can get the device object
// through the settings:
log.debug "settings.myswitch.currentSwitch: ${settings.myswitch.currentSwitch}"
...
```

284.1.71 state

A map of name/value pairs that SmartApps can use to save and retrieve data across SmartApp executions.

Signature: Map state

Returns: Map - a map of name/value pairs.

```
state.count = 0
state.count = state.count + 1

log.debug "state.count: ${state.count}"

// use array notation if you wish
log.debug "state['count']: ${state['count']}"

// you can store lists and maps to make more interesting structures
state.listOfMaps = [[key1: "vall", bool1: true],
                   [otherKey: ["string1", "string2"]]]
```

Warning: Though state can be treated as a map in most regards, certain convenience operations that you may be accustomed to in maps will not work with state. For example, `state.count++` will not increment the count - use the longer form of `state.count = state.count + 1`.

284.1.72 stringToMap()

Parses a comma-delimited string into a map.

Signature: Map stringToMap(String string)

Parameters: String string - A comma-delimited string to parse into a map.

Returns: Map - a map created from the comma-delimited string.

Example:

```
def testStr = "key1: value1, key2: value2"
def testMap = stringToMap(testStr)

log.debug "stringToMap: ${testMap}"
log.debug "stringToMap.key1: ${testMap.key1}" // => value1
log.debug "stringToMap.key2: ${testMap.key2}" // => value2
```

284.1.73 subscribe()

Subscribes to the various Events for a device or Location. The specified handlerMethod will be called when the Event is fired.

All event handler methods will be passed an *Event* (page 1017) that represents the Event causing the handler method to be called.

Signature: void subscribe(deviceOrDevices, String attributeName, handlerMethod)

```
void subscribe(deviceOrDevices, String attributeNameAndValue,
handlerMethod)
```

```
void subscribe(Location location, handlerMethod)
```

```
void subscribe(Location location, String eventName, handlerMethod)
```

```
void subscribe(app, handlerMethod)
```

Parameters: deviceOrDevices - The *Device* (page 1003) or list of devices to subscribe to.

String attributeName - The attribute to subscribe to.

String attributeNameAndValue - The specific attribute value to subscribe to, in the format "<attributeName>.<attributeValue>"

handlerMethod - The method to call when the Event is fired. Can be a String of the method name or the method reference itself.

Location (page 1036) location - The Location to subscribe to

app - Pass in the available app property in the SmartApp to subscribe to touch Events in the app.

Returns: void

Example:

```
preferences {
    section() {
        input "mycontact", "capability.contactSensor"
        input "myswitches", "capability.switch", multiple: true
    }
}
```

```
// subscribe to all state change Events for ``contact`` attribute of a contact sensor
subscribe(mycontact, "contact", handlerMethod)

// subscribe to all state changes for all switch devices configured
subscribe(myswitches, "switch", handlerMethod)

// subscribe to the "open" event for the contact sensor - only when the state changes to "open" will
subscribe(mycontact, "contact.open", handlerMethod)

// subscribe to all state change Events for the installed SmartApp's Location
subscribe(location, handlerMethod)

// subscribe to touch Events for this app - handlerMethod called when app is touched
subscribe(app, appTouchMethod)

// all event handler methods must accept an event parameter
def handlerMethod(evt) {
    log.debug "event name: ${evt.name}"
    log.debug "event value: ${evt.value}"
}
```

284.1.74 subscribeToCommand()

Subscribes to device commands that are sent to a device. The specified `handlerMethod` will be called whenever the specified command is sent.

Signature: `void subscribeToCommand(device, commandName, handlerMethod)`

Parameters:

- `device` - The *Device* (page 1003) to subscribe to.
- `String commandName` - The command to subscribe to
- `handlerMethod` - the method to call when the command is called.

Returns: void

Example:

```
preferences {
    section() {
        input "switch1", "capability.switch"
    }
}
...
subscribeToCommand(switch1, "on", onCommand)
...
// called when the on() command is called on switch1
def onCommand(evt) {...}
```

284.1.75 timeOfDayIsBetween()

Find if a given date is between a lower and upper bound.

Signature: Boolean `timeOfDayIsBetween(Date start, Date stop, Date value, TimeZone timeZone)`

Parameters: `Date start` - The start date to compare against.

`Date stop` - The end date to compare against.

`Date value` - The date to compare to start and stop.

`TimeZone timeZone` - The time zone for this comparison.

Returns: Boolean - `true` if the specified date is between the start and stop dates, false otherwise.

Example:

```
def between = timeOfDayIsBetween(new Date() - 1, new Date() + 1,
                                new Date(), location.timeZone)
log.debug "between: $between" => true
```

284.1.76 `timeOffset()`

Gets a time offset in milliseconds for the specified input.

Signature: Long `timeOffset(Number minutes)`

`Long timeOffset(String hoursAndMinutesString)`

Parameters: `Number minutes` - The number of minutes to get the offset in milliseconds for.

`String hoursAndMinutesString` - A string in the format of "hh:mm" to get the offset in milliseconds for. Negative offsets are specified by prefixing the string with a minus sign ("-02:30").

Returns: Long - the time offset in milliseconds for the specified input.

Example:

```
def off1 = timeOffset(24) // => 1440000
def off2 = timeOffset("2:30") // => 9000000
def off2again = timeOffset(150) // => 9000000
def off3 = timeOffset("-02:30") // => -9000000
```

284.1.77 `timeToday()`

Gets a `Date` object for today's date, for the specified time in the date-time parameter.

Signature: Date `timeToday(String timeString [, TimeZone timeZone])`

Parameters: `String timeString` - Either an ISO-8601 date string as returned from `time` input preferences, or a simple time string in "hh:mm" format ("21:34").

`TimeZone timeZone` (*optional*) - The time zone to use for determining the current day.

Warning: Although the `timeZone` argument is optional, it is *strongly encouraged* that you use it. Not specifying the `timeZone` results in the SmartThings platform trying to calculate the time zone based on the date and time zone offsets in the input string.

To avoid time zone errors, you should specify the `timeZone` argument (you can get the time zone from the `location` object: `location.timeZone`)

Future releases may remove the option to call `timeToday` without a time zone.

Returns: `Date` - the `Date` that represents today's date for the specified time.

Example:

```
preferences {
    section() {
        input "startTime", "time"
        input "endTime", "time"
    }
}
...
def start = timeToday(startTime, location.timeZone)
def end = timeToday(endTime, location.timeZone)
```

284.1.78 timeTodayAfter()

Returns a `Date` of the next occurrence of the time specified in the input, relative to a reference time.

Signature: `Date timeTodayAfter(String startTimeString, String timeString [, TimeZone timeZone])`

Parameters: `String startTimeString` - The reference time. Can be an ISO-8601 date string as returned from time input preferences, or a simple time string in "hh:mm" format ("21:34").

`String timeString` - The time string whose next occurrence is queried. Can be an ISO-8601 date string as returned from time input preferences, or a simple time string in "hh:mm" format ("21:34").

`TimeZone timeZone` (*optional*) - The time zone used for determining the current date and time.

Warning: Although the `timeZone` argument is optional, it is *strongly encouraged* that you use it. Not specifying the `timeZone` results in the SmartThings platform trying to calculate the time zone based on the date and time zone offsets in the input string.

To avoid time zone errors, you should specify the `timeZone` argument (you can get the time zone from the location object: `location.timeZone`)

Future releases may remove the option to call `timeToday` without a time zone.

Returns: `Date` - If time specified by `timeString` has already occurred prior to `startTimeString` then returns the next day `Date` object when the `timeString` time occurs next. If `timeString` time has not yet occurred relative to `startTimeString`, then returns today's `Date` object when the `timeString` time will occur. Since only the occurrence of `timeString` after the elapse of `startTimeString` time is considered, the `Date` returned is guaranteed to be later than the `startTimeString` date.

Example:

```
preferences {
    section() {
        input "time1", "time"
        input "time2", "time"
    }
}
...
// assume time1 entered as 20:20
// assume time2 entered as 14:05
// since 14:05 time today has already elapsed prior to 20:20 reference time today,
// the nextTime would be tomorrow's date, 14:05 time (the next occurrence of 14:05 time)
def nextTime = timeTodayAfter(time1, time2, location.timeZone)
```

284.1.79 timeZone()

Get a *TimeZone* object for the specified time value entered as a SmartApp preference. This will get the current time zone of the mobile app (not the Hub Location).

Signature: `TimeZone timeZone(String timePreferenceString)`

Parameters: `String timePreferenceString` - The time value string in ISO-8061 format as entered as input in SmartApp time preferences.

Returns: `TimeZone` - the `TimeZone` for the time value as specified by the `timePreferenceString`.

Example:

```
preferences {
    section() {
        input "mytime", "time"
    }
}
...
def enteredTimeZone = timeZone(mytime)
...
```

284.1.80 toDateTime()

Get a `Date` object for the specified string.

Signature: `Date toDateTime(dateTimeString)`

Parameters: `String dateTimeString` - the date-time string for which to get a `Date` object, in ISO-8061 format as used by time preferences

Returns: `Date` - the `Date` for the specified `dateTimeString`.

Example:

```
preferences {
    section() {
        input "mytime", "time"
    }
}
...
Date myTimeAsDate = toDateTime(mytime)
...
```

284.1.81 unschedule()

Deletes all scheduled jobs for the SmartApp. If using the optional `method` parameter, then it deletes the scheduled job for the specified handler name only.

Signature: `void unschedule(String method = '')`

Returns: `void`

Note: This can be an expensive operation if unscheduling all scheduled jobs; make sure you need to do this before calling. Typically called in the `updated()` (page 898) method if the SmartApp has set up recurring schedules.

284.1.82 unsubscribe()

Deletes all subscriptions for the installed SmartApp, or for a specific device or devices if specified.

Typically should be called in the `updated()` (page 898) method, since device preferences may have changed.

Signature: `unsubscribe([deviceOrDevices])`

Parameters: `deviceOrDevices` (*optional*) - The device or devices for which to unsubscribe from. If not specified, all subscriptions for this installed SmartApp will be deleted.

Returns: `void`

Example:

```
def updated() {
    unsubscribe()
}
```

284.2 Device Handler

Device Handlers are the virtual representation of a physical device.

A Device Handler defines a `metadata()` (page 964) method that defines the device's definition, UX information, as well as how it should behave in the IDE simulator.

A Device Handler typically also defines a `parse()` (page 948) method that is responsible for transforming raw messages from the device into Events for the SmartThings platform.

Device Handlers must also define methods for any supported commands, either through its supported capabilities, or device-specific commands.

For more information about the structure of Device Handlers, refer to the Device Handler's Guide.

Tip: Writing a Device Handler is considered a somewhat advanced topic. Understanding of how a Device Handler is organized and operates is assumed in this reference documentation. You should be familiar with the contents of the Device Handler's Guide to get the most out of this documentation.

Methods expected to be defined by Device Handlers:

284.2.1 <command name>()

Note: This method is expected to be defined by Device Handlers.

The definition for a Command supported by this Device Handler. Every Command that a Device Handler supports, either through its capabilities or custom commands, must have a corresponding command method defined.

Commands are the things that a device can do. For example, the “Switch” capability defines the commands “on” and “off”. Every Device that supports the “Switch” capability must define an implementation of these commands. This is done by defining methods with the name of the command. For example, `def on() {}` and `def off()`.

The exact implementation of a command method will vary greatly depending upon the device. The command method is responsible for sending protocol and device-specific commands to the physical device.

Signature: `Object <command name>([arguments])>`

Returns: `Object` - Commands may return any object, but typically do not return anything since they perform some type of action.

Example:

```
metadata {
    // Automatically generated. Make future change here.
    definition (name: "Centralite Switch", namespace: "smarththings", author: "SmartThings") {
        ...
        capability "Switch"
        ...
    }
    ...
    // capability "Switch" declared, so all supported commands
    // of "Switch" must be implemented:
    def on() {
        // device-specific commands to turn the switch on
    }

    def off() {
        // device-specific commands to turn the switch off
    }
    ...
}
```

284.2.2 parse()

Note: This method is expected to be defined by Device Handlers.

Called when messages from a device are received from the Hub. The parse method is responsible for interpreting those messages and returning *Event* (page 1017) definitions. Event definitions are maps that contain, at a minimum, name and value entries. They may also contain unit, displayText, displayed, isStateChange, and linkText entries if the default, automatically generated values of these Event properties are to be overridden. See the *createEvent()* (page 955) documentation for a description of these properties.

Because the `parse()` method is responsible for handling raw device messages, their implementations vary greatly across different Device Handlers.

The `parse()` method may return a map defining the *Event* (page 1017) to create and propagate through the SmartThings platform, or a list of Events if multiple Events should be created. It may also return a HubAction or list of HubAction objects in the case of LAN-connected devices.

Signature: `Map parse(String description)`

`List<Map> parse(String description)`

`HubAction parse(String description)`

`List<HubAction> parse(String description)`

Example:

```
def parse(String description) {
    log.debug "Parse description $description"
    def name = null
    def value = null
    if (description?.startsWith("read attr -")) {
        def descMap = parseDescriptionAsMap(description)
        log.debug "Read attr: $description"
        if (descMap.cluster == "0006" && descMap.attrId == "0000") {
            name = "switch"
            value = descMap.value.endsWith("01") ? "on" : "off"
        } else {
            def reportValue = description.split(",").find {it.split(":")[0].trim() == "value"?.split
            name = "power"
            // assume 16 bit signed for encoding and power divisor is 10
            value = Integer.parseInt(reportValue, 16).intdiv(10)
        }
    } else if (description?.startsWith("on/off:")) {
        log.debug "Switch command"
        name = "switch"
        value = description?.endsWith(" 1") ? "on" : "off"
    }

    // createEvent returns a Map that defines an Event
    def result = createEvent(name: name, value: value)
    log.debug "Parse returned ${result?.descriptionText}"

    // returning the Event definition map creates an Event
    // in the SmartThings platform, and propagates it to
    // SmartApps subscribed to the device events.
    return result
}
```

284.2.3 addChildDevice()

Adds a child device to a Device Handler. An example use is in a composite device Device Handler.

A parent may have multiple children, but only one level of children is allowed (i.e., if a device has a parent, it may not have children itself).

Warning: A parent may have at most 500 children.

Signature: `DeviceWrapper addChildDevice(String typeName, String deviceNetworkId, hubId, Map properties)`

```
DeviceWrapper addChildDevice(String namespace, String typeName, String
deviceNetworkId, hubId, Map properties)
```

Parameters: `String namespace` - the namespace for the device. If not specified, defaults to the namespace of the current Device Handler executing the call.

`String typeName` - the device type name

`String deviceNetworkId` - the device network id of the device

`hubId` - (optional) The hub id. Defaults to null

`Map properties` (optional) - A map with device properties. Available options are:

Option	Description
<code>isComponent</code>	Allowed values are <code>true</code> and <code>false</code> . When <code>true</code> hides the device from the Things view and doesn't let it be separately deleted. (Example: This value is <code>true</code> for the ZooZ ZEN 20 and <code>false</code> for Hue bridge.)
<code>componentName</code>	A way to refer to this particular child. It should be a Java Bean name (i.e. no spaces). It is used to refer to the device in the parent's detail view. This option is only needed when <code>isComponent</code> is <code>true</code> .
<code>componentLabel</code>	The plain-english name (or i18n key) to be used by the UX.
<code>completedSetup</code>	Specify <code>true</code> to complete the setup for the child device; <code>false</code> to have the user complete the installation. It should be <code>true</code> if <code>isComponent</code> is <code>true</code> . Defaults to <code>false</code> .
<code>label</code>	The label for the device.

Returns: `Device` (page 1003) - The device that was created.

Throws: `UnknownDeviceTypeException` - If a Device Handler with the specified name and namespace is not found.

`IllegalArgumentException` - If the `deviceNetworkId` is not specified.

`ValidationException` - If the this device already has a parent.

`SizeLimitExceededException` - If this device already has the maximum number of children allowed (500).

Example:

```
// on installation, create child devices
def installed() {
    createChildDevices()
}

def createChildDevices() {

    // This device (power strip) has five outlets
    for (i in 1..5) {
        // can omit namespace (first arg) if it is the same as this device
        addChildDevice("smarththings", "Zooz Power Strip Outlet", "${device.deviceNetworkId}-ep${i}",
            [completedSetup: true, label: "${device.displayName} (CH${i})",
             isComponent: true, componentName: "ch${i}", componentLabel: "Channel ${i}"])
    }
}
```

284.2.4 apiServerUrl()

Returns the URL of the server where this Device Handler can be reached for API calls, along with the specified path appended to it. Use this instead of hard-coding a URL to ensure that the correct server URL for this installed instance is returned.

Signature: `String apiServerUrl(String path)`

Parameters: `String path` - the path to append to the URL

Returns: The URL of the server for this installed instance of the Device Handler.

Example:

```
// logs <server url>/my/path
log.debug "apiServerUrl: ${apiServerUrl("/my/path")}"

// The leading "/" will be added if you don't specify it
// logs <server url>/my/path
log.debug "apiServerUrl: ${apiServerUrl("my/path")}"
```

284.2.5 attribute()

Called within the *definition()* (page 956) method to declare that this Device Handler supports an attribute not defined by any of its declared capabilities.

For any supported attribute, it is expected that the Device Handler creates and sends Events with the name of the attribute in the *parse()* (page 948) method.

Signature: `void attribute(String attributeName, String attributeType [, List possibleValues])`

Parameter: `String attributeName` - the name of the attribute

`String attributeType` - the type of the attribute. Available types are “string”, “number”, and “enum”

`List possibleValues` (*optional*) - the possible values for this attribute. Only valid with the “enum” attributeType.

Returns: void

Example:

```
metadata {
    definition (name: "Some Device Name", namespace: "somenamespace",
               author: "Some Author") {
        capability "Switch"
        capability "Polling"
        capability "Refresh"

        // also support the attribute "myCustomAttribute" - not defined by supported capabilities.
        attribute "myCustomAttribute", "number"

        // enum attribute with possible values "light" and "dark"
        attribute "someOtherName", "enum", ["light", "dark"]
    }
    ...
}
```

284.2.6 capability()

Called in the *definition()* (page 956) method to define that this device supports the specified capability.

Important: Whatever commands and attributes defined by that capability should be implemented by the Device Handler. For example, the “Switch” capability specifies support for the “switch” attribute and the “on” and “off” commands - any Device Handler supporting the “Switch” capability must define methods for the commands, and support the “switch” attribute by creating the appropriate Events (with the name of the attribute, e.g., “switch”)

Signature: void capability(String capabilityName)

Parameters: String capabilityName - the name of the capability. This is the long-form name of the Capability name, not the “preferences reference”.

Returns: void

Example:

```
metadata {
    definition (name: "Cerbco Light Switch", namespace: "lennyv62",
               author: "Len Veil") {
        capability "Switch"
        ...
    }
    ...
}

def parse(description) {
    // handle device messages, determine what value of the Event is
    return createEvent(name: "switch", value: someValue)
}

// need to define the on and off commands, since those
// are supported by "Switch" capability
def on() {
    ...
}

def off() {
}
```

284.2.7 carouselTile()

Called within the *tiles()* (page 979) method to define a tile often used in conjunction with the Image Capture capability, to allow users to scroll through recent pictures.

Signature: void carouselTile(String tileName, String attributeName [,Map options, Closure closure])

Parameters: String tileName - the name of the tile. This is used to identify the tile when specifying the tile layout.

String attributeName - the attribute this tile is associated with. Each tile is associated with an attribute of the device. The typical pattern is to prefix the attribute name with "device." - e.g., "device.water".

Map options (*optional*) - Various options for this tile. Valid options are found in the table below:

option	type	description
width	Integer	controls how wide the tile is. Default is 1.
height	Integer	controls how tall this tile is. Default is 1.
canChangeIcon	Boolean	true to allow the user to pick their own icon. Defaults to false.
canChangeBackground	Boolean	true to allow a user to choose their own background image for the tile. Defaults to false.
decoration	String	specify "flat" for the tile to render without a ring.
range	String	used to specify a custom range. In the form of "<lower bound>..<upper bound>"

Closure `closure` (*optional*) - a closure that defines any states for the tile.

Returns: void

Example:

```
tiles {
    carouselTile("cameraDetails", "device.image", width: 3, height: 2) {}
}
```

284.2.8 childDeviceTile()

Called within the `tiles()` (page 979) method in a parent Device Handler of a composite device to define the display of a child device tile. The mobile user interface of a composite parent device is built typically by combining tiles from multiple child devices.

Signature: void childDeviceTile(String tileName, String componentName [, Map options, Closure closure])

Returns: void

Parameters: String `tileName` - the name of the tile. This is used to identify the tile when specifying the tile layout.

String `componentName` - the name of the component child device. This name is the same as the `componentName` in the `addChildDevice()` in the composite parent Device Handler.

Map `options` (*optional*) - Various options for this tile. Valid options are found in the table below:

option	type	description
width	Integer	controls how wide the tile is. Default is 1.
height	Integer	controls how tall this tile is. Default is 1.
childTileName	String	name of the tile in the child Device Handler.

Closure `closure` (*optional*) - A closure that calls any `state()` (page 975) methods to define how the tile should appear for various attribute values.

Example:

```
metadata {
    definition (name: "Simulated Refrigerator", namespace: "smarthings/testing", author: "SmartThings") {
        capability "Contact Sensor"
    }
    tiles {
        childDeviceTile("mainDoor", "mainDoor", height: 2, width: 2, childTileName: "mainDoor")
    }
    ...
}
```

```
}  
def installed() {  
    state.counter = state.counter ? state.counter + 1 : 1  
    if (state.counter == 1) {  
        // A tile with the name "mainDoor" exists in the tiles() method of the child Device Handler  
        addChildDevice(  
            "Simulated Refrigerator Door",  
            "${device.deviceNetworkId}.2",  
            null,  
            [completedSetup: true, label: "${device.label} (Main Door)", componentName: "mainDoor",  
            ]  
        )  
    }  
}
```

284.2.9 command()

Called within the *definition()* (page 956) method to declare that this Device Handler supports a command not defined by any of its declared capabilities.

For any supported command, it is expected that the Device Handler define a *<command name>()* (page 948) method with a corresponding name.

Signature: void command(String commandName [, List parameterTypes])

Parameter: String commandName - the name of the command.

List parameterTypes (*optional*) - a list of strings that defines the types of the parameters the command requires (in order), if any. Typical values are “string”, “number”, and “enum”.

Returns: void

Example:

```
metadata {  
    definition (name: "Some Device Name", namespace: "somenamespace",  
                author: "Some Author") {  
        capability "Switch"  
        capability "Polling"  
        capability "Refresh"  
  
        // also support the attribute "myCustomCommand" - not defined by supported capabilities.  
        command "myCustomCommand"  
  
        // commands can take parameters  
        command "myCustomCommandWithParams", ["string", "number"]  
  
    }  
    ...  
}  
  
def myCustomCommand() {  
    ...  
}  
  
def myCustomCommandWithParams(def stringArg, def numArg) {  
    ...  
}
```

284.2.10 controlTile()

Called within the `tiles()` (page 979) method to define a tile that allows the user to input a value within a range. A common use case for a control tile is a light dimmer.

Signature: `void controlTile(String tileName, String attributeName, String controlType [, Map options, Closure closure])`

Returns: `void`

Parameters: `String tileName` - the name of the tile. This is used to identify the tile when specifying the tile layout.

`String attributeName` - the attribute this tile is associated with. Each tile is associated with an attribute of the device. The typical pattern is to prefix the attribute name with "device." - e.g., "device.water".

`String controlType` - the type of control. Either "slider" or "control".

`Map options` (*optional*) - Various options for this tile. Valid options are found in the table below:

option	type	description
width	Integer	controls how wide the tile is. Default is 1.
height	Integer	controls how tall this tile is. Default is 1.
canChangeIcon	Boolean	true to allow the user to pick their own icon. Defaults to false.
canChangeBackground	Boolean	true to allow a user to choose their own background image for the tile. Defaults to false.
decoration	String	specify "flat" for the tile to render without a ring.
range	String	used to specify a custom range. In the form of "<lower bound>..<upper bound>"

`Closure closure` (*optional*) - A closure that calls any `state()` (page 975) methods to define how the tile should appear for various attribute values.

Example:

```
tiles {
    controlTile("levelSliderControl", "device.level", "slider", height: 1,
               width: 2, inactiveLabel: false, range:"(0..100)") {
        state "level", action:"switch level.setLevel"
    }
}
```

284.2.11 createEvent()

Creates a Map that represents an `Event` (page 1017) object. Typically used in the `parse()` (page 948) method to define Events for particular attributes. The resulting map is then returned from the `parse()` method. The SmartThings platform will then create an Event object and propagate it through the system.

Signature: `Map createEvent(Map options)`

Parameters: `Map options` - The various properties that define this Event. The available options are listed below. It is not necessary, or typical, to define all the available options. Typically only the name and value options are required.

Property	Type	Description
name (required)	String	the name of the Event. Typically corresponds to an attribute name of a capability.
value (required)	Object	the value of the Event. The value is stored as a string, but you can pass numbers or other objects.
descriptionText	String	the description of this Event. This appears in the mobile application activity for the device. If not specified, this will be created using the Event name and value.
displayed	Boolean	specify <code>true</code> to display this Event in the mobile application activity feed, <code>false</code> to not display. Defaults to <code>true</code> .
linkText	String	name of the Event to show in the mobile application activity feed.
isState-Change	Boolean	specify <code>true</code> if this Event caused a device attribute to change state. Typically not used, since it will be set automatically.
unit	String	a unit string, if desired. This will be used to create the <code>descriptionText</code> if it (the <code>descriptionText</code> option) is not specified.
data	Map	A map of additional information to store with the Event

Example:

```
def parse(String description) {
    ...

    def evt1 = createEvent(name: "someName", value: "someValue")
    def evt2 = createEvent(name: "someOtherName", value: "someOtherValue")

    return [evt1, evt2]
}
```

284.2.12 definition()

Called within the `metadata()` (page 964) method, and defines some basic information about the device, as well as the supported capabilities, commands, and attributes.

Signature: `void definition(Map definitionData, Closure closure)`

Parameters: `Map definitionData` - defines various metadata about this Device Handler. Valid options are:

option	type	description
name	String	the name of this Device Handler
namespace	String	the namespace for this Device Handler. Typically the same as the author's github user name.
author	String	the name of the author.

`Closure closure` - A closure with method calls to `capability()` (page 952), `command()` (page 954), or `attribute()` (page 951).

Returns: void

Example:

```
metadata {
    definition (name: "My Device Name", namespace: "mynamespace",
               author: "My Name") {
        capability "Switch"
    }
}
```



```

    capability "Polling"
    capability "Refresh"

    command "someCustomCommand"

    attribute "someCustomAttribute", "number"
  }
  ...
}

```

284.2.13 details()

Used within the *tiles()* (page 979) method to define the order that the tiles should appear in.

Signature: void details(List<String> tileDefinitions)

Parameters: List<String>tileDefinitions - A list of tile names that defines the order of the tiles (left-to-right, top-to-bottom)

Returns: void

Example:

```

tiles {
    standardTile("switchTile", "device.switch", width: 2, height: 2,
        canChangeIcon: true) {
        state "off", label: '${name}', action: "switch.on",
            icon: "st.switches.switch.off", backgroundColor: "#ffffff"
        state "on", label: '${name}', action: "switch.off",
            icon: "st.switches.switch.on", backgroundColor: "#E60000"
    }
    valueTile("powerTile", "device.power", decoration: "flat") {
        state "power", label: '${currentValue} W'
    }
    standardTile("refreshTile", "device.power", decoration: "ring") {
        state "default", label: '', action: "refresh.refresh",
            icon: "st.secondary.refresh",
    }

    main "switchTile"

    // defines what order the tiles are defined in
    details(["switchTile", "powerTile", "refreshTile"])
}

```

284.2.14 device

The Device object, from which its current properties and history can be accessed. As of now this object is a different type than the Device object available in SmartApps. At some point these will be merged, but for now the properties and methods of the device object available to the Device Handler are discussed in the example below:

```
...
// Gets the most recent State for the given attribute
def state1 = device.currentState("someAttribute")
def state2 = device.latestState("someOtherAttribute")

// Gets the current value for the given attribute
// Return type will vary depending on the device
def curVal1 = device.currentValue("someAttribute")
def curVal2 = device.latestValue("someOtherAttribute")

// gets the display name of the device
def displayName = device.displayName

// gets the internal unique system identifier for this device
def thisId = device.id

// gets the internal name for this device
def thisName = device.name

// gets the user-defined label for this device
def thisLabel = device.label
```

284.2.15 fingerprint()

Called within the *definition()* (page 956) method to define the information necessary to pair this device to the Hub. See the Fingerprinting Section of the Device Handler guide for more information.

284.2.16 getApiServerUrl()

Returns the URL of the server where this Device Handler can be reached for API calls. Use this instead of hard-coding a URL to ensure that the correct server URL for this installed instance is returned.

Signature: `String getApiServerUrl()`

Returns: `String` - the URL of the server where this Device Handler can be reached.

284.2.17 getChildDevices()

Gets a list of all child devices for this device.

Signature: `List<ChildDeviceWrapper> getChildDevices()`

Returns: `List<Device` (page 1003)> - a list of child devices for this device

Example:

```
def children = getChildDevices()

log.debug "device has ${children.size()} children"
children.each { child ->
    log.debug "child ${child.displayName} has deviceNetworkId ${child.deviceNetworkId}"
}
```

284.2.18 getColorUtil()

Returns the *ColorUtilities* (page 1001) object.

Signature: `ColorUtilities getColorUtil()`

Returns: *ColorUtilities* (page 1001)

284.2.19 getImage()

Returns a `ByteArrayInputStream` for the image stored using *storeImage()* (page 976) or *storeTemporaryImage()* (page 977) with the specified name.

An exception is thrown if the requested image does not exist for this device.

Signature: `ByteArrayInputStream getImage(String name)`

Parameters: `String name` - The name of the image to retrieve.

Returns: `ByteArrayInputStream` - The input stream of bytes for this image.

Example:

```
ByteArrayInputStream imgStream = getImage("some-existing-image-name")
```

284.2.20 httpDelete()

Executes an HTTP DELETE request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

Signature: `void httpDelete(String uri, Closure closure)`

`void httpDelete(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP DELETE call to.

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Request content type and Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

Returns: `void`

284.2.21 httpGet()

Executes an HTTP GET request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpGet(String uri, Closure closure)`

`void httpGet(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP GET call to

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Request content type and Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure - closure` - The closure that will be called with the response of the request.

Example:

```
def params = [
    uri: "http://httpbin.org",
    path: "/get"
]

try {
    httpGet(params) { resp ->
        resp.headers.each {
            log.debug "${it.name} : ${it.value}"
        }
        log.debug "response contentType: ${resp.contentType}"
        log.debug "response data: ${resp.data}"
    } catch (e) {
        log.error "something went wrong: $e"
    }
}
```

284.2.22 httpHead()

Executes an HTTP HEAD request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

Signature: `void httpHead(String uri, Closure closure)`

`void httpHead(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP HEAD call to

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Request content type and Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure `closure` - The closure that will be called with the response of the request.

284.2.23 httpPost()

Executes an HTTP POST request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPost(String uri, String body, Closure closure)`

`void httpPost(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP GET call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Request content type and Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure `closure` - The closure that will be called with the response of the request.

Example:

```
try {
    httpPost("http://mysite.com/api/call", "id=XXX&value=YYY") { resp ->
        log.debug "response data: ${resp.data}"
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.debug "something went wrong: $e"
}
```

284.2.24 httpPostJson()

Executes an HTTP POST request with a JSON-encoded body and content type, and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPostJson(String uri, String body, Closure closure)`

`void httpPostJson(String uri, Map body, Closure closure)`

`void httpPostJson(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP POST call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Request content type and Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

Example:

```
def params = [
    uri: "http://postcatcher.in/catchers/<yourUniquePath>",
    body: [
        param1: [subparam1: "subparam 1 value",
                subparam2: "subparam2 value"],
        param2: "param2 value"
    ]
]

try {
    httpPostJson(params) { resp ->
        resp.headers.each {
            log.debug "${it.name} : ${it.value}"
        }
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.debug "something went wrong: $e"
}
```

284.2.25 httpPut()

Executes an HTTP PUT request and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPut(String uri, String body, Closure closure)`
`void httpPut(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP GET call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Request content type and Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

Example:

```
try {
    httpPut("http://mysite.com/api/call", "id=XXX&value=YYY") { resp ->
        log.debug "response data: ${resp.data}"
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.error "something went wrong: $e"
}
```

284.2.26 httpPutJson()

Executes an HTTP PUT request with a JSON-encoded body and content type, and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPutJson(String uri, String body, Closure closure)`
`void httpPutJson(String uri, Map body, Closure closure)`
`void httpPutJson(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP PUT call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Request content type and Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure *closure* - The closure that will be called with the response of the request.

284.2.27 main()

Used to define what tile appears on the main “Things” view in the mobile application. Can be called within the *tiles()* (page 979) method.

Signature: void main(String tileName)

Parameters: String tileName - the name of the tile to display as the main tile.

Returns: void

Example:

```
tiles {
    standardTile("switchTile", "device.switch", width: 2, height: 2,
        canChangeIcon: true) {
        state "off", label: '${name}', action: "switch.on",
            icon: "st.switches.switch.off", backgroundColor: "#ffffff"
        state "on", label: '${name}', action: "switch.off",
            icon: "st.switches.switch.on", backgroundColor: "#E60000"
    }
    valueTile("powerTile", "device.power", decoration: "flat") {
        state "power", label: '${currentValue} W'
    }
    standardTile("refreshTile", "device.power", decoration: "ring") {
        state "default", label: '', action: "refresh.refresh",
            icon: "st.secondary.refresh",
    }

    // The "switchTile" will be main tile, displayed in the "Things" view
    main "switchTile"
    details(["switchTile", "powerTile", "refreshTile"])
}
```

284.2.28 metadata()

Used to define metadata such as this Device Handler’s supported capabilities, attributes, commands, and UX information.

Signature: void metadata(Closure closure)

Parameters: Closure `closure` - a closure that defines the metadata. The closure is expected to have the following methods called in it: `definition()` (page 956), `simulator()` (page 973), and `tiles()` (page 979).

Returns: void

Example:

```
metadata {
    definition(name: "device name", namespace: "yournamespace", author: "your name") {
        // supported capabilities, commands, attributes,
    }
    simulator {
        // simulator metadata
    }
    tiles {
        // tiles metadata
    }
}
```

284.2.29 reply()

Called in the `simulator()` (page 973) method to model the behavior of a physical device when a virtual instance of the Device Handler is run in the IDE.

The simulator matches command strings generated by the device to those specified in the `commandString` argument of a reply method and, if a match is found, calls the Device Handler's parse method with the corresponding `messageDescription`.

For example, the reply method `reply "2001FF,2502": "command: 2503, payload: FF"` models the behavior of a physical Z-Wave switch in responding to an Basic Set command followed by a Switch Binary Get command. The result will be a call to the parse method with a Switch Binary Report command with a value of 255, i.e., the turning on of the switch. Modeling turn off would be done with the reply method `reply "200100,2502": "command: 2503, payload: 00"`.

Signature: void `reply(String commandString, String messageDescription)`

Parameters: `String commandString` - a String that represents the command.

`String messageDescription` - a String that represents the message description.

Returns: void

Example:

```
metadata {
    ...

    // simulator metadata
    simulator {
        // 'on' and 'off' will appear in the messages dropdown, and send
        // "on/off: 1 to the parse method"
        status "on": "on/off: 1"
        status "off": "on/off: 0"

        // simulate reply messages from the device
        reply "zcl on-off on": "on/off: 1"
        reply "zcl on-off off": "on/off: 0"
```

```
}  
...  
}
```

284.2.30 runEvery1Minute()

Creates a recurring schedule that executes the specified `handlerMethod` every minute. Using this method will pick a random start time in the next minute, and run every minute after that.

Signature: `void runEvery1Minute(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the 1 minute period.

Parameters: `handlerMethod` - The method to call every minute. Can be the name of the method as a string, or a reference to the method.

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: `void`

Example:

```
runEvery1Minute(handlerMethod1)  
runEvery1Minute(handlerMethod2, [data: [key1: 'vall']])  
  
def handlerMethod1() {  
    log.debug "handlerMethod1"  
}  
  
def handlerMethod2(data) {  
    log.debug "handlerMethod2, data: $data"  
}
```

284.2.31 runEvery5Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every five minutes. Using this method will pick a random start time in the next five minutes, and run every five minutes after that.

Signature: `void runEvery5Minutes(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the 5 minute period.

Parameters: `handlerMethod` - The method to call every five minutes. Can be the name of the method as a string, or a reference to the method.

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: `void`

Example:

```
runEvery5Minutes(handlerMethod1)
runEvery5Minutes(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.2.32 runEvery10Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every ten minutes. Using this method will pick a random start time in the next ten minutes, and run every ten minutes after that.

Signature: `void runEvery10Minutes(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the ten minute period.

Parameters: `handlerMethod` - The method to call every ten minutes. Can be the name of the method as a string, or a reference to the method.

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: `void`

Example:

```
runEvery10Minutes(handlerMethod1)
runEvery10Minutes(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.2.33 runEvery15Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every fifteen minutes. Using this method will pick a random start time in the next five minutes, and run every five minutes after that.

Signature: `void runEvery15Minutes(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the fifteen minute period.

Parameters: `handlerMethod` - The method to call every fifteen minutes. Can be the name of the method as a string, or a reference to the method.

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: `void`

Example:

```
runEvery15Minutes(handlerMethod1)
runEvery15Minutes(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.2.34 runEvery30Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every thirty minutes. Using this method will pick a random start time in the next thirty minutes, and run every thirty minutes after that.

Signature: `void runEvery30Minutes(handlerMethod[, options])`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the thirty minute period.

Parameters: `handlerMethod` - The method to call every thirty minutes. Can be the name of the method as a string, or a reference to the method.

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: `void`

Example:

```
runEvery30Minutes(handlerMethod1)
runEvery30Minutes(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.2.35 runEvery1Hour()

Creates a recurring schedule that executes the specified handlerMethod every hour. Using this method will pick a random start time in the next hour, and run every hour after that.

Signature: void runEvery1Hour(handlerMethod[, options])

Tip: This is preferred over using schedule(cronExpression, handlerMethod) for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the one hour period.

Parameters: handlerMethod- The method to call every hour. Can be the name of the method as a string, or a reference to the method.

options (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
data	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
runEvery1Hour(handlerMethod1)
runEvery1Hour(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.2.36 runEvery3Hours()

Creates a recurring schedule that executes the specified handlerMethod every three hours. Using this method will pick a random start time in the next hour, and run every three hours after that.

Signature: void runEvery3Hours(handlerMethod[, options])

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the three hour period.

Parameters: `handlerMethod` - The method to call every three hours. Can be the name of the method as a string, or a reference to the method.

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: `void`

Example:

```
runEvery3Hours(handlerMethod1)
runEvery3Hours(handlerMethod2, [data: [key1: 'val1']])

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2(data) {
    log.debug "handlerMethod2, data: $data"
}
```

284.2.37 runIn()

Executes a specified `handlerMethod` after `delaySeconds` have elapsed.

Signature: `void runIn(delayInSeconds, handlerMethod [, options])`

Tip: It's important to note that we will attempt to run this method at this time, but cannot guarantee exact precision. We typically expect per-minute level granularity, so if using with values less than sixty seconds, your mileage will vary.

Parameters: `delayInSeconds` - The number of seconds to execute the `handlerMethod` after.

`handlerMethod` - The method to call after `delayInSeconds` has passed. Can be a string or a reference to the method.

`options` (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
<code>overwrite</code>	<code>true</code> or <code>false</code>	Specify [<code>overwrite: false</code>] to not overwrite any existing pending schedule handler for the given method (the default behavior is to overwrite the pending schedule). Specifying [<code>overwrite: false</code>] can lead to multiple different schedules for the same handler method, so be sure your handler method can handle this.
<code>data</code>	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
runIn (300, myHandlerMethod)
runIn (400, "myOtherHandlerMethod", [data: [flag: true]])

def myHandlerMethod() {
    log.debug "handler method called"
}

def myOtherHandlerMethod(data) {
    log.debug "other handler method called with flag: $data.flag"
}
```

284.2.38 runOnce()

Executes the handlerMethod once at the specified date and time.

Signature: void runOnce(dateTime, handlerMethod [, options])

Parameters: dateTime - When to execute the handlerMethod. Can be either a [Date](#) object or an ISO-8601 date string. For example, `new Date() + 1` would run at the current time tomorrow, and `"2017-07-04T12:00:00.000Z"` would run at noon GMT on July 4th, 2017.

handlerMethod - The method to execute at the specified dateTime. This can be a reference to the method, or the method name as a string.

options (*optional*) - A map of parameters, with the following keys supported:

Key	Possible values	Description
overwrite	true or false	Specify <code>[overwrite: false]</code> to not overwrite any existing pending schedule handler for the given method (the default behavior is to overwrite the pending schedule). Specifying <code>[overwrite: false]</code> can lead to multiple different schedules for the same handler method, so be sure your handler method can handle this.
data	A map of data	A map of data that will be passed to the handler method.

Returns: void

Example:

```
// execute handler at 4 PM CST on October 21, 2015 (e.g., Back to the Future 2 Day!)
runOnce ("2015-10-21T16:00:00.000-0600", handler)

def handler() {
    ...
}
```

284.2.39 schedule()

Creates a scheduled job that calls the handlerMethod once per day at the time specified, or according to a cron schedule.

Signature: void schedule(dateTime, handlerMethod)
void schedule(cronExpression, handlerMethod)

Parameters:

- dateTime - A [Date](#) object, an ISO-8601 formatted date time string.
- String cronExpression - A cron expression that specifies the schedule to execute on.
- handlerMethod - The method to call. This can be a reference to the method itself, or the method name as a string.

Returns: void

Tip: Since calling schedule () with a dateTime argument creates a recurring scheduled job to execute *every day* at the specified time, the *date information is ignored. Only the time portion of the argument is used.*

Tip: Full documentation for the cron expression format can be found in the [Quartz Cron Trigger Tutorial](#)

Example:

```
preferences {
    section() {
        input "timeToRun", "time"
    }
}

...
// call handlerMethod1 at time specified by user input
schedule(timeToRun, handlerMethod1)

// call handlerMethod2 every day at 3:36 PM CST
schedule("2015-01-09T15:36:00.000-0600", handlerMethod2)

// execute handlerMethod3 every hour on the half hour (using a randomly chosen seconds field)
schedule("12 30 * * * ?", handlerMethod3)
...

def handlerMethod1 () {...}
def handlerMethod2 () {...}
def handlerMethod3 () {...}
```

284.2.40 sendEvent()

Create and fire an [Event](#) (page 1017) . Typically a Device Handler will return the map returned from [createEvent\(\)](#) (page 955) , which will allow the platform to create and fire the Event. In cases where you need to fire the Event (outside of the [parse\(\)](#) (page 948) method), sendEvent () is used.

Signature: void sendEvent(Map properties)

Parameters: Map properties - The properties of the Event to create and send.

Here are the available properties:

Property	Description
name (required)	<code>String</code> - The name of the Event. Typically corresponds to an attribute name of a capability.
value (required)	The value of the Event. The value is stored as a string, but you can pass numbers or other objects.
descriptionText	<code>String</code> - The description of this Event. This appears in the mobile application activity for the device. If not specified, this will be created using the Event name and value.
displayed	Pass <code>true</code> to display this Event in the mobile application activity feed, <code>false</code> to not display. Defaults to <code>true</code> .
linkText	<code>String</code> - Name of the Event to show in the mobile application activity feed.
isStateChange	<code>true</code> if this Event caused a device attribute to change state. Typically not used, since it will be set automatically.
unit	<code>String</code> - a unit string, if desired. This will be used to create the <code>descriptionText</code> if it (the <code>descriptionText</code> option) is not specified.
data	A map of additional information to store with the Event

Tip: Not all Event properties need to be specified. ID properties like `deviceId` and `locationId` are automatically set, as are properties like `isStateChange`, `displayed`, and `linkText`.

Returns: void

Example:

```
sendEvent(name: "temperature", value: 72, unit: "F")
```

284.2.41 simulator()

Defines information used to simulate device interaction in the IDE. Can be called in the `metadata()` (page 964) method.

Signature: void simulator(Closure closure)

Parameters: Closure closure - the closure that defines the `status()` (page 976) and `reply()` (page 965) messages.

Returns: void

Example:

```
metadata {
    ...

    // simulator metadata
    simulator {
        // 'on' and 'off' will appear in the messages dropdown, and send
        // "on/off: 1 to the parse method"
        status "on": "on/off: 1"
        status "off": "on/off: 0"

        // simulate reply messages from the device
        reply "zcl on-off on": "on/off: 1"
        reply "zcl on-off off": "on/off: 0"
    }
}
```

```
}
    ...
}
```

284.2.42 standardTile()

Called within the `tiles()` (page 979) method to define a tile to display current state information. For example, to show that a switch is on or off, or that there is or is not motion.

Signature: `void standardTile(String tileName, String attributeName [, Map options, Closure closure])`

Returns: `void`

Parameters: `String tileName` - the name of the tile. This is used to identify the tile when specifying the tile layout.

`String attributeName` - the attribute this tile is associated with. Each tile is associated with an attribute of the device. The typical pattern is to prefix the attribute name with "device." - e.g., "device.water".

`Map options` (*optional*) - Various options for this tile. Valid options are found in the table below:

option	type	description
width	Integer	controls how wide the tile is. Default is 1.
height	Integer	controls how tall this tile is. Default is 1.
canChangeIcon	Boolean	true to allow the user to pick their own icon. Defaults to false.
canChangeBackground	Boolean	true to allow a user to choose their own background image for the tile. Defaults to false.
decoration	String	specify "flat" for the tile to render without a ring.

`Closure closure` (*optional*) - A closure that calls any `state()` (page 975) methods to define how the tile should appear for various attribute values.

Example:

```
tile {
    standardTile("water", "device.water", width: 2, height: 2) {
        state "dry", icon: "st.alarm.water.dry", backgroundColor: "#ffffff"
        state "wet", icon: "st.alarm.water.wet", backgroundColor: "#53a7c0"
    }
}
```

284.2.43 state

A map of name/value pairs that a Device Handler can use to save and retrieve data across executions.

Signature: `Map state`

Returns: `Map` - a map of name/value pairs.

```
state.count = 0
state.count = state.count + 1

log.debug "state.count: ${state.count}"
```

```
// use array notation if you wish
log.debug "state['count']: ${state['count']}"

// you can store lists and maps to make more interesting structures
state.listOfMaps = [[key1: "vall", bool1: true],
                    [otherKey: ["string1", "string2"]]]
```

Warning: Though `state` can be treated as a map in most regards, certain convenience operations that you may be accustomed to in maps will not work with `state`. For example, `state.count++` will not increment the count - use the longer form of `state.count = state.count + 1`.

284.2.44 state()

Called within any of the various tiles method's closure to define options to be used when the current value of the tile's attribute matches the value argument.

Signature: `void state(stateName, Map options)`

Parameters: `String stateName` - the name of the attribute value for which to display this state for.

`Map options` - a map that defines additional information for this state. The valid options are:

option	type	description
action	String	the action to take when this tile is pressed. The form is <code><capabilityReference>.<command></code> .
back-ground-Color	String	a hexadecimal color code to use for the background color. This has no effect if the tile has decoration: "flat".
back-ground-Colors	String	specify a list of maps of attribute values and colors. The mobile app will match and interpolate between these entries to select a color based on the value of the attribute.
default-State	Boolean	specify true if this state should be the active state displayed for this tile.
icon	String	the identifier of the icon to use for this state. You can view the icon options here.
label	String	the label for this state.

Returns: `void`

Example:

```
...
standardTile("water", "device.water", width: 2, height: 2) {
    // when the "water" attribute has the value "dry", show the
    // specified icon and background color
    state "dry", icon:"st.alarm.water.dry", backgroundColor:"#ffffff"

    // when the "water" attribute has the value "wet", show the
    // specified icon and background color
    state "wet", icon:"st.alarm.water.wet", backgroundColor:"#53a7c0"
}

valueTile("temperature", "device.temperature", width: 2, height: 2) {
    state("temperature", label:'${currentValue}°',
        backgroundColors:[
            [value: 31, color: "#153591"],
```

```
[value: 44, color: "#1e9cbb"],
[value: 59, color: "#90d2a7"],
[value: 74, color: "#44b621"],
[value: 84, color: "#f1d801"],
[value: 95, color: "#d04e00"],
[value: 96, color: "#bc2323"]
]
)
}
...

```

284.2.45 status()

The status method is called in the *simulator()* (page 973) method, and populates the select box that appears under virtual devices in the IDE. Can be called in the *simulator()* (page 973) method.

Signature: void status(String name, String messageDescription)

Parameters: *String* name - any unique string and is used to refer to this status message in the select box.

String messageDescription - should be a parseable message for this Device Handler. It's passed to the Device Handler's parse method when select box entry is sent in the simulator. For example, status "on": "command: 2003, payload: FF" will send a Z-Wave Basic Report command to the Device Handler's parse method when the on option is selected and sent.

Returns: void

Example:

```
metadata {
    ...

    // simulator metadata
    simulator {
        // 'on' and 'off' will appear in the messages dropdown, and send
        // "on/off: 1 to the parse method"
        status "on": "on/off: 1"
        status "off": "on/off: 0"

        // simulate reply messages from the device
        reply "zcl on-off on": "on/off: 1"
        reply "zcl on-off off": "on/off: 0"
    }
    ...
}
```

284.2.46 storeImage()

Stores an image represented by a [ByteArrayInputStream](#) and emits an Event with name "image".

storeImage() is often in used in conjunction with the *carouselTile()* (page 952) and cloud-connected camera devices to store and display images.

JPEG and PNG image formats are supported.

Note: Images stored using `storeImage()` are stored for 365 days, after which they will be permanently deleted.

The `carouselTile()` can display images for the past seven days.

Signature: `void storeImage(String name, ByteArrayInputStream is, String contentType = "image/jpeg")` throws `Exception`

Parameters: `String name` - The name associated with the image, consisting of alphanumeric, `'_'`, `'-'`, and `'.'` characters. This name must be unique per device instance, and should not include the file extension.

`ByteArrayInputStream is` - The input stream of bytes representing the image. The total size may not exceed 1 megabyte.

`String contentType (optional)` - The content type of the image. Optional, and defaults to `"image/jpeg"`. Other supported values are `"image/jpg"` and `"image/png"`.

Throws: `InvalidParameterException` if the name does not solely consist of alphanumeric, `'_'`, `'-'`, and `'.'` characters.

`InvalidParameterException` if the size of total bytes to be stored exceeds one megabyte.

`Exception` - if the current Device Handler execution is attempting to store more than two images.

Example:

```
def params = [
    uri: "http://static.tvtropes.org/pmwiki/pub/images/catsbeard_9105.jpg"
]

try {
    httpGet(params) { response ->
        if (response.status == 200 && response.headers.'Content-Type'.contains("image/jpeg")) {
            def imageBytes = response.data
            if (imageBytes) {
                state.imgCount = state.imgCount + 1
                def name = "test${state.imgCount}"

                // the response data is already a ByteArrayInputStream, no need to convert
                try {
                    storeImage(name, imageBytes)
                } catch ( ) {
                    log.error "error storing image: $e"
                }
            }
        }
    }
} catch (err) {
    log.error ("Error making request: $err")
}
```

284.2.47 storeTemporaryImage()

Transfers an image temporarily stored via a *HubAction* (page 1029) request to a LAN-connected camera device to longer-lasting storage, and emits an event with name `"image"`. Typically used in conjunction with the *carouselTile()* (page 952) to store and display images captured by a camera device.

Only the JPEG image format is supported.

Note: Images stored using `storeTemporaryImage()` are stored for 365 days, after which they will be permanently deleted.

The `carouselTile()` can display images for the past seven days.

Signature: `void storeTemporaryImage(String key, String name)`

Parameters: `String key` - The key for this image, extracted from the response map sent to the `parse()` (page 948) method.

`String name` - The name associated with the image, consisting of alphanumeric, `'_'`, `'-'`, and `'.'` characters. This name must be unique per device instance, and should not include the file extension.

Throws: `InvalidParameterException` if the name does not solely consist of alphanumeric, `'_'`, `'-'`, and `'.'` characters.

Example:

```
// take() command method from the Image Capture capability
def take() {
    def host = getHostAddress()
    def port = host.split(":")[1]

    def path = "/some/path/"

    def hubAction = new physicalgraph.device.HubAction(
        method: "GET",
        path: path,
        headers: [HOST:host]
    )

    // outputMsgToS3: true required to store this image temporarily!
    hubAction.options = [outputMsgToS3:true]

    return hubAction
}

/**
 * Utility method to get the host addresses
 */
private getHostAddress() {
    def parts = device.deviceNetworkId.split(":")
    def ip = convertHexToIP(parts[0])
    def port = convertHexToInt(parts[1])
    return ip + ":" + port
}

def parse(String description) {

    def map = stringToMap(description)

    // if the message has the tempImageKey, we know it's a response from
    // an image stored via the HubAction. Need to move it to longer-lasting
    // storage with storeTemporaryImage()
    if (map.tempImageKey) {
        try {
            storeTemporaryImage(map.tempImageKey, getPictureName())
        } catch (Exception e) {
            log.error e
        }
    }
}
```

```

    } else if (map.error) {
        log.error ("Error: ${map.error}")
    }

    // parse other messages too
}

/**
 * Utility method to get a unique picture name
 */
private getPictureName() {
    return java.util.UUID.randomUUID().toString().replaceAll('-', '')
}

```

284.2.48 tiles()

Defines the user interface for the device in the mobile app. It's composed of one or more *standardTile()* (page 974) , *valueTile()* (page 979) , *carouselTile()* (page 952) , or *controlTile()* (page 955) methods, as well as a *main()* (page 964) and *details()* (page 957) method.

Signature: void tiles(Closure closure)

Parameters: Closure closure - A closure that defines the various tiles and metadata.

Returns: void

Example:

```

tiles {
    standardTile("switchTile", "device.switch", width: 2, height: 2,
        canChangeIcon: true) {
        state "off", label: '${name}', action: "switch.on",
            icon: "st.switches.switch.off", backgroundColor: "#ffffff"
        state "on", label: '${name}', action: "switch.off",
            icon: "st.switches.switch.on", backgroundColor: "#E60000"
    }
    valueTile("powerTile", "device.power", decoration: "flat") {
        state "power", label: '${currentValue} W'
    }
    standardTile("refreshTile", "device.power", decoration: "ring") {
        state "default", label: '', action: "refresh.refresh",
            icon: "st.secondary.refresh",
    }

    main "switchTile"
    details(["switchTile", "powerTile", "refreshTile"])
}

```

284.2.49 valueTile()

Defines a tile that displays a specific value. Typical examples include temperature, humidity, or power values. Called within the *tiles()* (page 979) method.

Signature: void valueTile(String tileName, String attributeName [, Map options, Closure closure])

Returns: void

Parameters: `String tileName` - the name of the tile. This is used to identify the tile when specifying the tile layout.

`String attributeName` - the attribute this tile is associated with. Each tile is associated with an attribute of the device. The typical pattern is to prefix the attribute name with "device." - e.g., "device.power".

`Map options` (*optional*) - Various options for this tile. Valid options are found in the table below:

option	type	description
width	Integer	controls how wide the tile is. Default is 1.
height	Integer	controls how tall this tile is. Default is 1.
canChangeIcon	Boolean	true to allow the user to pick their own icon. Defaults to false.
canChangeBackground	Boolean	true to allow a user to choose their own background image for the tile. Defaults to false.
decoration	String	specify "flat" for the tile to render without a ring.

`Closure closure` (*optional*) - A closure that calls any *state()* (page 975) methods to define how the tile should appear for various attribute values.

Example:

```
tiles {
    valueTile("power", "device.power", decoration: "flat") {
        state "power", label: '${currentValue} W'
    }
}
```

284.2.50 zigbee

A utility class for parsing and formatting ZigBee messages.

Signature: Zigbee zigbee

Returns: A reference to the *ZigBee utility class* (page 1046).

284.2.51 zwave

The utility class for parsing and formatting Z-Wave command messages.

Signature: ZWave zwave

Returns: A reference to the ZWave helper class. See the *Z-Wave Reference* (page 1059) for more information.

Example:

```
// On command implementation for a Z-Wave switch
def on() {
    delayBetween ([
        zwave.basicV1.basicSet (value: 0xFF).format (),
        zwave.switchBinaryV1.switchBinaryGet ().format ()
    ])
```



```

    })
}

```

284.3 AppState

The AppState object encapsulates information about the state of a SmartApp attribute. These attributes are usually set in SmartApps using the sendEvent method. Here is a small code snippet that illustrates a potential use case.

```

// If the current state of the "status" attribute is not what is expected,
// then send an event to update it.
if (app.currentState("status")?.value != "expectedValue") {
    def text = "$app.label someAction"
    sendEvent(name: "status", value: "expectedValue", linkText: app.label,
              descriptionText: text, eventType:"SOLUTION_EVENT", data: [icon: icon, backgroundColor: color])
}

```

284.3.1 getDateValue()

The value of this Event, if the value can be parsed to a Date.

Signature: Date getDateValue()

Returns: Date - the value of this Event as a Date.

Warning: getDateValue() will throw an Exception if the value of the Event is not parseable to a Date. You should wrap calls in a try/catch block.

Example:

```

def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    // get the value of this event as an Double
    // throws an exception if the value is not convertible to a Date
    try {
        log.debug "The dateValue of this event is ${myState.dateValue}"
        log.debug "myState.dateValue instanceof Date? ${myState.dateValue instanceof Date}"
    } catch (e) {
        log.debug("Trying to get the dateValue for ${myState.name} threw an exception", e)
    }
}

```

284.3.2 getId()

The unique system identifier for this Event.

Signature: String getId()

Returns: String - the unique device identifier for this Event.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    log.debug "event id: ${myState.id}"
}
```

284.3.3 getDescriptionText()

The description of the Event that is to be displayed to the user in the mobile application.

Signature: String getDescriptionText()

Returns: String - the description of this Event to be displayed to the user in the mobile application.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    log.debug "event description text: ${myState.descriptionText}"
}
```

284.3.4 getDoubleValue()

The value of this Event, if the value can be parsed to a Double.

Signature: Double getDoubleValue()

Returns: Double - the value of this Event as a Double.

Warning: getDoubleValue() will throw an Exception if the value of the Event is not parseable to a Double. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    // get the value of this event as a Double
    // throws an exception if the value is not convertible to a Double
    try {
        log.debug "The doubleValue of this event is ${myState.doubleValue}"
        log.debug "myState.doubleValue instanceof Double? ${myState.doubleValue instanceof Double}"
    } catch (e) {
        log.debug("Trying to get the doubleValue for ${myState.name} threw an exception", e)
    }
}
```

284.3.5 getFloatValue()

The value of this Event as a Float, if it can be parsed into a Float.

Signature: Float getFloatValue()

Returns: Float - the value of this Event as a Float.

Warning: getFloatValue() will throw an Exception if the Event's value is not parseable to a Float. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    // get the value of this event as an Float
    // throws an exception if not convertable to Float
    try {
        log.debug "The floatValue of this event is ${myState.floatValue}"
        log.debug "myState.floatValue instanceof Float? ${myState.floatValue instanceof Float}"
    } catch (e) {
        log.debug("Trying to get the floatValue for ${myState.name} threw an exception", e)
    }
}
```

284.3.6 getIntegerValue()

The value of this Event as an Integer.

Signature: Integer getIntegerValue()

Returns: Integer - the value of this Event as an Integer.

Warning: getIntegerValue() throws an Exception of the Event value cannot be parsed to an Integer. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    // get the value of this event as an Integer
    // throws an exception if not convertable to Integer
    try {
        log.debug "The integerValue of this event is ${myState.integerValue}"
        log.debug "The integerValue of this event is an Integer: ${myState.integerValue instanceof Integer}"
    } catch (e) {
        log.debug("Trying to get the integerValue for ${myState.name} threw an exception", e)
    }
}
```

284.3.7 getIsoDate()

Acquisition time of this Event as an ISO-8601 String.

Signature: `String getIsoDate()`

Returns: `String` - The acquisition time of this Event as an ISO-8601 String.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    log.debug "event isoDate: ${myState.isoDate}"
}
```

284.3.8 getJsonValue()

Value of the Event as a parsed JSON data structure.

Signature: `Object getJsonValue()`

Returns: `Object` - The value of the Event as a JSON structure

Warning: `getJsonValue()` throws an Exception if the value of the Event cannot be parsed into a JSON object.

You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    // get the value of this event as a JSON structure
    // throws an exception if the value is not convertible to JSON
    try {
        log.debug "The jsonValue of this event is ${myState.jsonValue}"
    } catch (e) {
        log.debug("Trying to get the jsonValue for ${myState.name} threw an exception", e)
    }
}
```

284.3.9 getLastUpdated()

The last time this Event was updated as a Date.

Signature: `Date getLastUpdated()`

Returns: `Date` - The last time this Event was updated as a Date.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    log.debug "event was last updated: ${myState.lastUpdated}"
}
```

284.3.10 getLongValue()

The value of this Event as a Long.

Signature: Long getLongValue()

Returns: Long - the value of this Event as a Long.

Warning: getLongValue() throws an Exception if the value of the Event cannot be parsed to a Long. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    // get the value of this event as an Long
    // throws an exception if not convertible to Long
    try {
        def evtLongValue = myState.longValue
        log.debug "The longValue of this event is $evtLongValue"
        log.debug "evt.longValue instanceof Long? ${evtLongValue instanceof Long}"
    } catch ( ) {
        log.debug("Trying to get the longValue for ${myState.name} threw an exception", e)
    }
}
```

284.3.11 getName()

The name of this Event.

Signature: String getName()

Returns: String - the name of this Event.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    log.debug "the name of this event: ${myState.name}"
}
```

284.3.12 getNumberValue()

The value of this Event as a Number.

Signature: Number getNumberValue()

Returns: Number - the value of this Event as a BigDecimal.

Warning: getNumberValue() throws an Exception if the value of the Event cannot be parsed to a Number. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    // get the value of this event as an Number
    // throws an exception if the value is not convertable to a Number
    try {
        def evtNumberValue = myState.numberValue
        log.debug "The numberValue of this event is ${evtNumberValue}"
        log.debug "evt.numberValue instanceof BigDecimal? ${evtNumberValue instanceof Number}"
    } catch ( ) {
        log.debug("Trying to get the numberValue for ${myState.name} threw an exception", e)
    }
}
```

284.3.13 getNumericValue()

The value of this Event as a Number.

Signature: Number getNumericValue()

Returns: Number - the value of this Event as a BigDecimal.

Warning: getNumericValue() throws an Exception if the value of the Event cannot be parsed to a Number. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    // get the value of this event as an Number
    // throws an exception if the value is not convertable to a BigDecimal
    try {
        def evtNumberValue = myState.numericValue
        log.debug "The numericValue of this event is ${evtNumberValue}"
        log.debug "evt.numericValue instanceof Number? ${evtNumberValue instanceof Number}"
    } catch ( ) {
        log.debug("Trying to get the numericValue for ${myState.name} threw an exception", e)
    }
}
```

284.3.14 getUnit()

The unit of measure for this Event, if applicable.

Signature: String getUnit()

Returns: String - the unit of measure of this Event, if applicable. null otherwise.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    log.debug "The unit for this event: ${myState.unit}"
}
```

284.3.15 getValue()

The value of this Event as a String.

Signature: String getValue()

Returns: String - the value of this Event as a String.

Example:

```
def eventHandler(evt) {
    def myState = app.currentState("someAttribute")
    log.debug "The value of this event as a string: ${myState.getValue()}"
}
```

284.3.16 getXyzValue()

Value of the Event as a 3-entry Map with keys 'x', 'y', and 'z' with BigDecimal values. For example:

```
[x: 1001, y: -23, z: -1021]
```

Typically only useful for getting position data from the "Three Axis" Capability.

Signature: Map<String, BigDecimal> getXyzValue()

Returns: Map < String , BigDecimal > - A map representing the X, Y, and Z coordinates.

Warning: getXyzValue() throws an Exception if the value of the Event cannot be parsed to an X-Y-Z data structure.

You should wrap calls in a try/catch block.

Example:

```
def positionChangeHandler(evt) {
    def myState = app.currentState("someAttribute")
    // get the value of this event as a 3 entry map with keys
    // 'x', 'y', 'z', and BigDecimal values
    // throws an exception if the value is not convertible to a Date
    try {
        log.debug "The xyzValue of this event is ${myState.xyzValue}"
        log.debug "myState.xyzValue instanceof Map? ${myState.xyzValue instanceof Map}"
    } catch ( ) {
        log.debug("Trying to get the xyzValue for ${myState.name} threw an exception", )
    }
}
```

284.4 Async HTTP API (Beta)

Beta Feature

The ability to make asynchronous HTTP requests is currently available as a beta development feature.

All beta asynchronous HTTP APIs exist in the `asynhttp_v1 namespace` (page 374). Approximately 30 days after the launch of this beta feature, we will evaluate metrics and your feedback, and make adjustments as necessary.

When released generally, it is likely that the v1 postfix will be dropped, and a deprecation period will be announced to change existing usages accordingly.

If, for unexpected reasons, usage of asynchronous HTTP requests has negative impacts on the SmartThings platform, SmartThings reserves the right to alter or remove any impacted asynchronous HTTP APIs without notice. This is highly unlikely and every effort will be made to avoid such a scenario.

If you experience issues or have feedback on these asynchronous HTTP APIs, please share them on [this community thread](#).

All asynchronous HTTP APIs are only available after including the “asynhttp_v1” API:

```
include 'asynhttp_v1'

def initialize() {
    // invoke methods on the injected asynhttp_v1 object that was included
    asynhttp_v1.get(...)
}
```

This documentation is specific to making requests using the Async HTTP API. For reference documentation on working with the response, see the [AsyncResponse \(Beta\)](#) (page 993) documentation.

284.4.1 delete()

Make a DELETE request which will not block execution and therefore can run longer than the execution timeout.

Signature: void delete(String callbackMethod = null, Map params, Map data = null)

Parameters:

String callbackMethod - the name of the method to call with the response. If null the response will be discarded after the request is made.

Map params - parameters for the request. Supported keys below:

Key	Description
uri (required)	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
request-Content-Type	The value of the Content-Type request header. Defaults to 'application/json'.
content-Type	The value of the Accept request header. Defaults to the value of the requestContentType parameter if not specified.
body	The request body to send. Can be a string, or if the requestContentType is "application/json", a Map or List (will be serialized to JSON).

Map data (*optional*) - A map of data to pass to the response handler.

Example:


```
include 'asynhttp_v1'

def initialize() {
  def params = [
    uri: 'https://someapi.com',
    path: '/some/path',
    body: [key1: 'value 1']
  ]
  asynhttp_v1.delete(processResponse, params)
}

def processResponse(response, data) { ... }
```

284.4.2 get()

Make a GET request which will not block execution and therefore can run longer than the execution timeout.

Signature: void get(String callbackMethod = null, Map params, Map data = null)

Parameters:

String callbackMethod - the name of the method to call with the response. If null the response will be discarded after the request is made.

Map params - parameters for the request. Supported keys below:

Key	Description
uri (required)	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
requestContentType	The value of the Content-Type request header. Defaults to 'application/json'.
contentType	The value of the Accept request header. Defaults to the value of the requestContentType parameter if not specified.

Map data (*optional*) - A map of data to pass to the response handler.

Example:

```
include 'asynhttp_v1'

def initialize() {
  def params = [
    uri: 'https://api.github.com',
    path: '/search/code',
    query: [q: "httpGet+repo:SmartThingsCommunity/SmartThingsPublic"],
    contentType: 'application/json'
  ]
  asynhttp_v1.get(processResponse, params)
}

def processResponse(response, data) { ... }
```

284.4.3 head()

Make a HEAD request which will not block execution and therefore can run longer than the execution timeout.

Signature: void head(String callbackMethod = null, Map params, Map data = null)

Parameters:

String callbackMethod - the name of the method to call with the response. If null the response will be discarded after the request is made.

Map params - parameters for the request. Supported keys below:

Key	Description
uri (required)	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
requestContentType	The value of the Content-Type request header. Defaults to 'application/json'.
contentType	The value of the Accept request header. Defaults to the value of the requestContentType parameter if not specified.

Map data (*optional*) - A map of data to pass to the response handler.

Example:

```
include 'asynhttp_v1'

def initialize() {
    def params = [
        uri: 'https://someapi.com',
        path: '/some/path',
        query: [key1: 'value 1']
    ]
    asynhttp_v1.head(processResponse, params)
}

def processResponse(response, data) { ... }
```

284.4.4 patch()

Make a PATCH request which will not block execution and therefore can run longer than the execution timeout.

Signature: void patch(String callbackMethod = null, Map params, Map data = null)

Parameters:

String callbackMethod - the name of the method to call with the response. If null the response will be discarded after the request is made.

Map params - parameters for the request. Supported keys below:

Key	Description
uri (required)	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
request-Content-Type	The value of the Content-Type request header. Defaults to 'application/json'.
content-Type	The value of the Accept request header. Defaults to the value of the requestContentType parameter if not specified.
body	The request body to send. Can be a string, or if the requestContentType is "application/json", a Map or List (will be serialized to JSON).

Map data (*optional*) - A map of data to pass to the response handler.

Example:

```
include 'asynhttp_v1'

def initialize() {
  def params = [
    uri: 'https://someapi.com',
    path: '/some/path',
    body: [key1: 'value 1']
  ]
  asynhttp_v1.patch(processResponse, params)
}

def processResponse(response, data) { ... }
```

284.4.5 post()

Make a POST request which will not block execution and therefore can run longer than the execution timeout.

Signature: void post(String callbackMethod = null, Map params, Map data = null)

Parameters:

String callbackMethod - the name of the method to call with the response. If null the response will be discarded after the request is made.

Map params - parameters for the request. Supported keys below:

Key	Description
uri (required)	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
request-Content-Type	The value of the Content-Type request header. Defaults to 'application/json'.
content-Type	The value of the Accept request header. Defaults to the value of the requestContentType parameter if not specified.
body	The request body to send. Can be a string, or if the requestContentType is "application/json", a Map or List (will be serialized to JSON).

Map data (*optional*) - A map of data to pass to the response handler.

Example:

```
include 'asynhttp_v1'

def initialize() {
  def params = [
    uri: 'https://someapi.com',
    path: '/some/path',
    body: [key1: 'value 1']
  ]
  asynhttp_v1.post(processResponse, params)
}

def processResponse(response, data) { ... }
```

284.4.6 put()

Make a PUT request which will not block execution and therefore can run longer than the execution timeout.

Signature: void put(String callbackMethod = null, Map params, Map data = null)

Parameters:

String callbackMethod - the name of the method to call with the response. If null the response will be discarded after the request is made.

Map params - parameters for the request. Supported keys below:

Key	Description
uri (required)	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
request-Content-Type	The value of the <code>Content-Type</code> request header. Defaults to <code>'application/json'</code> .
content-Type	The value of the <code>Accept</code> request header. Defaults to the value of the <code>requestContentType</code> parameter if not specified.
body	The request body to send. Can be a string, or if the <code>requestContentType</code> is <code>"application/json"</code> , a <code>Map</code> or <code>List</code> (will be serialized to JSON).

Map data (*optional*) - A map of data to pass to the response handler.

Example:

```
include 'asynhttp_v1'

def initialize() {
    def params = [
        uri: 'https://someapi.com',
        path: '/some/path',
        body: [key1: 'value 1']
    ]
    asynhttp_v1.put(processResponse, params)
}

def processResponse(response, data) { ... }
```

284.5 AsyncResponse (Beta)

Beta Feature

The ability to make asynchronous HTTP requests is currently available as a beta development feature.

All beta asynchronous HTTP APIs exist in the `asynhttp_v1` *namespace* (page 374). Approximately 30 days after the launch of this beta feature, we will evaluate metrics and your feedback, and make adjustments as necessary.

When released generally, it is likely that the `v1` postfix will be dropped, and a deprecation period will be announced to change existing usages accordingly.

If, for unexpected reasons, usage of asynchronous HTTP requests has negative impacts on the SmartThings platform, SmartThings reserves the right to alter or remove any impacted asynchronous HTTP APIs without notice. This is highly unlikely and every effort will be made to avoid such a scenario.

If you experience issues or have feedback on these asynchronous HTTP APIs, please share them on [this community thread](#).

The `AsyncResponse` object represents the response of an asynchronous HTTP request. An instance of it is passed to the response handler specified when making the request.

This documentation is specific to handling responses from asynchronous HTTP requests. For reference documentation regarding making the request, see the *Async HTTP API (Beta)* (page 987) documentation.

284.5.1 getData()

Return the response as a string. Throws an exception if the request failed to get a response (e.g. Connection timeout, Response timeout), or if the status code was not 2XX.

Signature: String getData()

Example:

```
def responseHandler(response, data) {
    log.debug "raw response: $response.data"
}
```

284.5.2 getErrorData()

In the Event of an error response, returns the response body as a string. Throws an exception if the response is successful and has a 2XX response.

Signature: String getErrorData()

Example:

```
def responseHandler(response, data) {
    if (response.hasError()) {
        log.debug "raw response: $response.errorData"
    }
}
```

284.5.3 getErrorJson()

If the response has an error, parses the response body as JSON and returns the corresponding data structure. Throws an exception if the response is successful and has a 2XX response, or if the body fails to parse as JSON.

Signature: JSONElement getErrorJson()

Example:

```
def responseHandler(response, data) {
    if (response.hasError()) {
        try {
            log.debug "error json: $response.errorJson"
        } catch (e) {
            log.debug "error parsing json - raw error data is $response.errorData"
        }
    }
}
```

284.5.4 getErrorMessage()

Gets a human-readable error message if the request failed to get a response (e.g. Connection timeout, Response timeout), or if the status code was not 2XX.

Signature: String getErrorMessage()

Example:

```
def responseHandler(response, data) {
    if (response.hasError()) {
        log.debug "error on response: $response.errorMessage"
    }
}
```

284.5.5 getErrorXml()

If the response has an error, parses the response body as XML and returns the corresponding data structure. Throws an exception if the response is successful and has a 2XX response, or if the body fails to parse as XML.

Note: You can learn more about Groovy XML parsing and GPath [here](#).

Signature: GPathResult getErrorXml()

Example:

```
def responseHandler(response, data) {
    if (response.hasError()) {
        try {
            def xml = response.errorXml
        } catch (-) {
            log.warn "could not parse body to XML"
        }
    }
}
```

284.5.6 getHeaders()

Get the headers of the response.

Signature: Map<String, String> getHeaders()

Returns: Map <String, String> - A map of response headers keyed by the header name.

Example:

```
def responseHandler(response, data) {
    def headers = response.headers
    headers.each {header, value ->
        log.debug "$header: $value"
    }
}
```

284.5.7 getJson()

Parses the response body as JSON and returns the corresponding data structure. Throws an exception if the body fails to parse as JSON, if the request failed to get a response (e.g. Connection timeout, Response timeout), or if the status code was not 2XX).

Signature: `JSONElement getJson()`

Example:

```
include 'asynhttp_v1'

def initialize() {
    def params = [
        uri: 'https://someapi.com',
        path: '/some/path',
        requestContentType: 'application/json'
    ]
    asynhttp_v1.get(processResponse, params)
}

def processResponse(response, data) {
    try {
        log.debug "json response is: $response.json"
    } catch (e) {
        log.error("exception during response processing", e)
    }
}
```

284.5.8 getStatus()

Get the status code of the response.

Signature: `int getStatus()`

Example:

```
def responseHandler(response, data) {
    log.debug "response status code is: $response.status"
}
```

284.5.9 getWarningMessages()

Gets a list of warning messages, if applicable. Returns an empty list if there are no warning messages.

Typically used for debugging purposes. For example, a warning message will be found if the response is larger than the allowable limit.

Signature: `List<String> getWarningMessages()`

Example:


```
def responseHandler(response, data) {
    log.debug "warning messages: ${response.warningMessages}"
}
```

284.5.10 getXml()

Parses the response body as XML and returns the corresponding data structure. Throws an exception if the body fails to parse as XML, if the request failed to get a response (e.g. Connection timeout, Response timeout), or if the status code was not 2XX).

Note: You can learn more about Groovy XML parsing and GPath [here](#).

Signature: GPathResult getXml()

Example:

```
def responseHandler(response, data) {
    if (!response.hasError()) {
        try {
            def xml = response.xml
        } catch (e) {
            log.warn "could not parse body to XML"
        }
    }
}
```

284.5.11 hasError()

Return if the request has an error of some sort. This will be `true` if the request failed to complete or returned a non-2XX status code, and `false` if the request succeeded with a 2XX status code.

Signature: boolean hasError()

Example:

```
def responseHandler(response, data) {
    if (response.hasError()) {
        log.error "response has error: ${response.getErrorMessage()}"
    }
}
```

284.6 Attribute

An Attribute represents specific information about the state of a device. For example, the “Temperature Measurement” capability has an attribute named “temperature” that represents the temperature data.

The Attribute object contains metadata information about the Attribute itself - its name, data type, and possible values.

You will typically interact with Attributes values directly, for example, using the `current<Uppercase attribute name>` (page 1005) method on a `Device` (page 1003) instance. That will get the *value* of the Attribute, which is typically what SmartApps are most interested in.

You can get the supported Attributes of a Device through the Device's `getSupportedAttributes()` (page 1011) method.

Warning: Referring to an Attribute directly from a Device by calling `someDevice.getAttributeName()` will return an Attribute object with only the `name` property available. This is available for legacy purposes only, and will likely be removed at some time.

To get a reference to an Attribute object, you should use the `getSupportedAttributes()` method on the Device object, and then find the desired Attribute in the returned List.

You can view the available attributes for all Capabilities in our [Capabilities Reference](#) (page 655).

284.6.1 `getDataType()`

Gets the data type of this Attribute.

Signature: `String getDataType()`

Returns: `String` - the data type of this Attribute. Possible types are “STRING”, “NUMBER”, “VECTOR3”, “ENUM”.

Example:

```
preferences {
    section() {
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def attrs = thetemp.supportedAttributes
attrs.each {
    log.debug "${thetemp.displayName}, attribute ${it.name}, dataType: ${it.dataType}"
}
...
```

284.6.2 `getName()`

The name of the Attribute.

Signature: `String getName()`

Returns: `String` - the name of this attribute

Example:

```
preferences {
    section() {
        input "myswitch", "capability.switch"
    }
}
...
// switch capability has an attribute named "switch"
```

```
def switchAttr = myswitch.switch
log.debug "switch attribute name: ${switchAttr.name}"
...
```

284.6.3 getValues()

The possible values for this Attribute, if the data type is “ENUM”.

Signature: List<String> getValues()

Returns: List<String> - the possible values for this Attribute, if the data type is “ENUM”. An empty list is returned if there are no possible values or if the data type is not “ENUM”.

Example:

```
preferences {
    section() {
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def attrs = thetemp.supportedAttributes
attrs.each {
    log.debug "${thetemp.displayName}, attribute ${it.name}, values: ${it.values}"
    log.debug "${thetemp.displayName}, attribute ${it.name}, dataType: ${it.dataType}"
}
...
```

284.7 Capability

The Capability object encapsulates information about a certain Capability.

A Capability object cannot be created. You can get the Capabilities for a given device using the capabilities method on a *Device* (page 1003) instance:

```
def capabilities = mydevice.capabilities
```

For documentation for the available Capabilities, you can refer to the *Capabilities Reference* (page 655).

284.7.1 getAttributes()

Signature: List<Attribute> getAttributes()

Returns: List<Attribute (page 997)> - A list of Attributes of this capability. An empty list will be returned if this Capability has no Attributes.

Example:

```
preferences {
    section() {
        input "mySwitch", "capability.switch"
    }
}
...
def mySwitchCaps = mySwitch.capabilities

// log each capability supported by the "mySwitch" device, along
// with all its supported attributes
mySwitchCaps.each {cap ->
    log.debug "Capability name: ${cap.name}"
    cap.attributes.each {attr ->
        log.debug "-- Attribute name: ${attr.name}"
    }
}
...
```

284.7.2 getCommands()

Signature: List<Command> getCommands()

Returns: List<Command> (page 1002) - A list of Commands of this capability. An empty list will be returned if this Capability has no commands.

Example:

```
preferences {
    section() {
        input "mySwitch", "capability.switch"
    }
}
...
def mySwitchCaps = mySwitch.capabilities

// log each capability supported by the "mySwitch" device, along
// with all its supported commands
mySwitchCaps.each {cap ->
    log.debug "Capability name: ${cap.name}"
    cap.commands.each {comm ->
        log.debug "-- Command name: ${comm.name}"
    }
}
...
```

284.7.3 getName()

The name of the capability.

Signature: String getName()

Returns: String - the name of the capability.

Example:

```

preferences {
    section() {
        input "mySwitch", "capability.switch"
    }
}
...
def mySwitchCaps = mySwitch.capabilities

// log each capability supported by the "mySwitch" device
mySwitchCaps.each {cap ->
    log.debug "Capability name: ${cap.name}"
}
...

```

284.8 ColorUtilities

Provides conversion utilities for working with different color representations. Every SmartApp and Device Handler can get a reference to the `ColorUtilities` class using the `getColorUtil()` method (or the shorthand property reference `colorUtil`, if you prefer).

```

def deepSkyBlueInHex = colorUtil.rgbToHex(0, 191, 255)
log.debug "RGB 0,191,255 in Hex is $deepSkyBlueInHex"

```

The *ColorUtilities* class works with RGB and hex color values. A full discussion of web colors is beyond the scope of this document, but the basic definitions used for SmartThings development are defined below.

RGB (Red, Green, Blue) The RGB (Red, Green, Blue) color model “is an additive color model in which red, green, and blue light are added together in various ways to reproduce a broad array of colors.”¹

HEX “A hex triplet is a six-digit, three-byte hexadecimal number used in HTML, CSS, SVG, and other computing applications to represent colors. The bytes represent the red, green and blue components of the color. One byte represents a number in the range 00 to FF (in hexadecimal notation), or 0 to 255 in decimal notation.”²

284.8.1 hexToRgb()

Converts a hex color string to RGB. Assumes the hex value is three or six characters in length, and may or may not include the leading “#” character.

Signature: `static List hexToRgb(String hex)`

Parameters: `String hex` - The hex color string to convert

Returns: `List` - The RGB color representation, ordered as `[red, green, blue]`

Example:

```

def skyBlueInRgb = colorUtil.hexToRgb('#00BFFF')
log.debug "sky blue in RGB: $skyBlueInRgb"

```

¹ Wikipedia: https://en.wikipedia.org/wiki/RGB_color_model

² Wikipedia: https://en.wikipedia.org/wiki/Web_colors

284.8.2 rgbToHex()

Converts an RGB value to a hexadecimal color string.

Signature: `static String rgbToHex(red, green, blue)` throws `IllegalArgumentException`

Parameters: `Integer red` - The red value, between 0 and 255

`Integer green` - The green value, between 0 and 255

`Integer blue` - The blue value, between 0 and 255

Returns: `String` - The hexadecimal representation of the RGB value

Throws: `IllegalArgumentException` - An `IllegalArgumentException` is thrown if any of the RGB values are not within the 0 to 255 range.

Example:

```
def deepskyblueInHex = colorUtil.rgbToHex(0, 191, 255)
log.debug "RGB 0,191,255 in Hex is $deepSkyBlueInHex"
```

284.9 Command

A Command represents an action you can perform on a Device.

An instance of a Command object encapsulates information about that Command. You cannot create a Command object; you can retrieve them from a *Capability* (page 999) or from a *Device* (page 1003):

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...

// Get a list of Commands supported by theswitch:
def switchCommands = theswitch.supportedCommands
log.debug "switchCommands: $switchCommands"

// Iterate through the supported capabilities, log all supported commands:
// commands property available via the Capability object
def caps = theswitch.capabilities
caps.commands.each {comm ->
    log.debug "-- Command name: ${comm.name}"
}
```

284.9.1 getArguments()

The list of argument types for the command.

Signature: `List<String> getArguments()`

Returns: `List < String >` - A list of the argument types for this command. One of `"STRING"`, `"NUMBER"`, `"VECTOR3"`, or `"ENUM"`.

Example:

```

preferences {
    section() {
        input "theSwitchLevel", "capability.switchLevel"
    }
}
...
def supportedCommands = theSwitchLevel.supportedCommands

// logs each command's arguments
supportedCommands.each {
    log.debug "arguments for swithLevel command ${it.name}: ${it.arguments}"
}
...

```

284.9.2 getName()

The name of the command.

Signature: String getName()

Returns: String - the name of this command.

Example:

```

preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def supportedCommands = theswitch.supportedCommands

// logs each command name supported by theswitch
supportedCommands.each {
    log.debug "command name: ${it.name}"
}
...

```

284.10 Device

The Device object represents a physical device in a SmartApp. When a user installs a SmartApp, they typically will select the devices to be used by the SmartApp. SmartApps can then interact with these Device objects to get device information, or send commands to the Device.

Device objects cannot be instantiated, but are created by the SmartThings platform and available via the name given in the preferences definition of a SmartApp:

```

preferences {
    section() {
        // prompt user to select a device that supports the switch capability.
        // assign the chosen device to a variable named "theswitch"
        input "theswitch", "capability.switch"
    }
}

```

```
}  
...  
// access Device instance using the input name:  
def deviceDisplayName = theswitch.displayName  
...
```

Note: Event history is limited to the last seven days. Methods that query devices for Event history will only query the last seven days. This will be called out in those methods, but is good to be generally aware of.

284.10.1 <attribute name>State

The latest *State* (page 1040) instance for the specified Attribute.

The exact name will vary depending on the device and its available attributes.

For example, the Thermostat capability supports several attributes. To get the State for any of the attributes, simply use the attribute name to construct the call. Consider the case of the “temperature” and “heatingSetpoint” attributes:

```
somethermostat.temperatureState  
somethermostat.heatingSetpointState
```

Signature: `State <attribute name>State`

Returns: *State* (page 1040) - The latest State instance for the specified Attribute.

Example:

```
preferences {  
    section() {  
        input "thetemp", "capability.temperatureMeasurement"  
    }  
}  
...  
// The Temperature Measurement has a "temperature" attribute.  
// so the form is <attribute name>State = temperatureState  
def tempState = thetemp.temperatureState  
...
```

284.10.2 <command name>()

Executes the specified command on the Device.

The method name will vary on the Device and Command being called.

For example, a Device that supports the Switch capability has both the `on()` and `off()` commands.

Some commands may take parameters; you will pass those parameters to the command as well.

Signature: `void <command name>()`

`void <command name>([delay: Number])`

`void <command name>(arguments)`

`void <command name>(arguments, [delay: Number'])`

Parameters: `arguments` - The arguments to the command, if required.

`Map options` - A map of options to send to the command. Only the `delay` option is currently supported:

option	type	description
<code>delay</code>	Number	The number of milliseconds to wait before sending the command to the device.

Returns: `void`

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thethermostat", "capability.thermostat"
    }
}
...
// call the "on" command on theswitch - no arguments
theswitch.on()

// call the "setHeatingSetpoint" command on thethermostat - takes an argument:
thethermostat.setHeatingSetpoint(72)

// A map specifying command options can be specified as the last parameter.
// Only supported options are "delay":
theswitch.on([delay: 30000]) // send command after 30 seconds
thethermostat.setHeatingSetpoint(72, [delay: 30000])
...
```

284.10.3 `current<Uppercase attribute name>`

The latest reported values for the specified attribute.

The specific signature will vary depending on the attribute name. Follow the pattern of `current` plus the attribute name, with the *first letter capitalized*.

For example, the Carbon Monoxide Detector capability has an attribute "carbonMonoxide". To get the latest value for this attribute, you would call:

```
def currentCarbon = someDevice.currentCarbonMonoxide
```

Signature: `Object current<Uppercase attribute name>`

Returns: `Object` - the latest reported values for the specified attribute. The specific type of object returned will vary depending on the specific attribute.

Tip: The exact returned type for various attributes depends upon the underlying capability and Device Handler.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
```

```
...
def switchattr = theswitch.currentSwitch
def tempattr = thetemp.currentTemperature

log.debug "current switch: $switchattr"
log.debug "current temp: $tempattr"

// switch attribute returned as a string
log.debug "switchattr instanceof String? ${switchattr instanceof String}"

// temperature attribute returned as a Number
log.debug "tempattr instanceof Number? ${tempattr instanceof Number}"
...
```

284.10.4 currentState()

Gets the latest *State* (page 1040) for the specified attribute.

Signature: `State currentState(String attributeName)`

Parameters: `String attributeName` - The name of the attribute to get the State for.

Returns: *State* (page 1040) - The latest State instance for the specified attribute.

Example:

```
preferences {
    section() {
        input "temp", "capability.temperatureMeasurement"
    }
}
...
def tempState = temp.currentState("temperature")
log.debug "state value: ${tempState.value}"
...
```

284.10.5 currentValue()

Gets the latest reported values of the specified attribute.

Signature: `Object currentValue(String attributeName)`

Parameters: `String attributeName` - The name of the attribute to get the latest values for.

Returns: *Object* - The latest reported values of the specified attribute. The exact return type will vary depending upon the attribute.

Warning: The exact returned type for various attributes is not adequately documented at this time. Until they are, we recommend that you save often and experiment, or even look at the specific Device Handler for the device you are working with.

Example:

```

preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def switchattr = theswitch.currentValue("switch")
def tempattr = thetemp.currentValue("temperature")

log.debug "current switch: $switchattr"
log.debug "current temp: $tempattr"

// switch attribute returned as a string
log.debug "switchattr instanceof String? ${switchattr instanceof String}"

// temperature attribute returned as a Number
log.debug "tempattr instanceof Number? ${tempattr instanceof Number}"
...

```

284.10.6 events()

Get a list of Events for the Device in reverse chronological order (newest first).

Note: Only Events in the last seven days will be returned via the `events()` method.

Signature: `List<Event> events([max: N])`

Parameters: `Map` options (*optional*) - Options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return. By default, the maximum is 10.

Returns: `List<Event>` (page 1017) - A list of Events in reverse chronological order (newest first).

Example:

```

def theEvents = someDevice.events()
def mostRecent20Events = someDevice.events(max: 20)

```

284.10.7 eventsBetween()

Get a list of Events between the specified start and end dates.

Note: Only Events from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero Events.

Signature: `List<Event> eventsBetween(Date startDate, Date endDate [, Map options])`

Parameters: `Date startDate` - the lower Date range for the query.

`Date endDate` - the upper Date range for the query.

`Map options` (*optional*) - Options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return. By default, the maximum is 10.

Returns: `List <:ref:event_ref>` - a list of Events between the specified start and end dates.

Example:

```
// 3 days ago
def startDate = new Date() - 3

// today
def endDate = new Date()

def theEvents = someDevice.eventsBetween(startDate, endDate)
log.debug "there were ${theEvents.size()} events in the last three days"

// events in the last 3 days - maximum of 5 events
def limitedEvents = someDevice.eventsBetween(startDate, endDate, [max: 5])
```

284.10.8 eventsSince()

Get a list of Events since the specified date.

Note: Only Events from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero Events.

Signature: `List<Event> eventsSince(Date startDate [, Map options])`

Parameters: `Date startDate` - the date to start the query from.

`Map options` (*optional*) - options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return. By default, the maximum is 10.

Returns: `List <Event` (page 1017)> - a list of Events since the specified date.

Example:

```
def eventsSinceYesterday = someDevice.eventsSince(new Date() - 1)
log.debug "there have been ${eventsSinceYesterday.size()} since yesterday"
```

284.10.9 getCapabilities()

The List of Capabilities provided by this Device.

Signature: `List<Capability> getCapabilities()`

Returns: `List <Capability` (page 999)> - a List of Capabilities supported by this Device.

Example:

```
def supportedCaps = someDevice.capabilities
supportedCaps.each {cap ->
    log.debug "This device supports the ${cap.name} capability"
}
```

284.10.10 getDeviceNetworkId()

Gets the device network ID for the device.

Signature: String getDeviceNetworkId()

Returns: String - the network ID for the device

284.10.11 getDisplayName()

The label of the Device assigned by the user.

Signature: String getDisplayName()

Returns: String - the label of the Device assigned by the user, null if no label is set.

Example:

```
def devLabel = someDevice.displayName
if (devLabel) {
    log.debug "label set by user: $devLabel"
} else {
    log.debug "no label set by user for this device"
}
```

284.10.12 getHub()

The Hub associated with this Device.

Signature: Hub getHub()

Returns: Hub (page 1027) - the Hub for this Device.

Example:

```
log.debug "Hub: ${someDevice.hub.name}"
```

284.10.13 getId()

The unique system identifier for this Device.

Signature: String getId()

Returns: String - the unique system identifier for this Device.

284.10.14 getLabel()

The name of the Device set by the user in the mobile application or Web IDE.

Signature: `String getLabel()`

Returns: `String` - the name of the Device as configured by the user.

284.10.15 getLastActivity()

The date of the last Event with a source of `device`. (i.e. not commands)

Signature: `String getLastActivity()`

Returns: `Date` - Date of the last Event with a source of `device`.

284.10.16 getManufacturerName()

Gets the manufacturer name of the device, as specified in the Device Handler's *fingerprint* (page 471). If the device was joined using a generic fingerprint, it is whatever the device reported while joining.

Not applicable for cloud or LAN-connected devices (`null` will be returned).

Signature: `String getManufacturerName()`

Returns: `String` - the manufacturer name of the device, or `null`.

Example:

```
preferences {
    input "switches", "capability.switch", multiple: true
}

def installed() {
    switches.each {
        log.debug "switch id: ${it.id}, manufacturer name: ${it.getManufacturerName()}"
    }
}
```

284.10.17 getModelName()

Gets the model name of the device, as specified in the Device Handler's *fingerprint* (page 471). If the device was joined using a generic fingerprint, it is whatever the device reported while joining.

Not applicable for cloud or LAN-connected devices (`null` will be returned).

Signature: `String getModelName()`

Returns: `String` - the model name of the device, or `null`.

Example:

```

preferences {
    input "switches", "capability.switch", multiple: true
}

def installed() {
    switches.each {
        log.debug "switch id: ${it.id}, model name: ${it.getModelName()}"
    }
}

```

284.10.18 getStatus()

Get the current status of the Device. If no status is found then `INACTIVE` is returned.

Signature: `String getStatus()`

Returns: `String` - the status of the Device or `INACTIVE` if one doesn't exist.

284.10.19 getName()

The internal name of the Device. Typically assigned by the system and editable only by a user in the IDE.

Signature: `String getName()`

Returns: `String` - the internal name of the Device.

284.10.20 getSupportedAttributes()

The list of *Attribute* (page 997) s for this Device.

Signature: `List<Attribute> getSupportedAttributes()`

Returns: `List<Attribute>` (page 997) - the list of Attributes for this Device. Includes both capability attributes as well as Device-specific attributes.

Example:

```

preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def theAtts = theswitch.supportedAttributes
theAtts.each { att ->
    log.debug "Supported Attribute: ${att.name}"
}
...

```

284.10.21 getSupportedCommands()

The list of *Command* (page 1002) s for this Device.

Signature: List<Command> getSupportedCommands()

Returns: List <*Command* (page 1002)> - the list of Commands for this Device. Includes both capability commands as well as Device-specific commands.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def theCommands = theswitch.supportedCommands
theCommands.each {com ->
    log.debug "Supported Command: ${com.name}"
}
...
```

284.10.22 getTypeName()

The type of the device.

Signature: String getTypeName()

Returns: String - the type of the device.

284.10.23 hasAttribute()

Determine if this Device has the specified attribute.

Tip: Attribute names are case-sensitive.

Signature: Boolean hasAttribute(String attributeName)

Parameters: String attributeName - the name of the attribute to check if the Device supports.

Returns: Boolean - true if this Device has the specified attribute. Returns a non-true value if not (may be null).

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def hasTempAttr = thetemp.hasAttribute("temperature")
// true, since this device supports the 'temperature' capability
```



```

log.debug "${thetemp.displayName} has temperature attribute? $hasTempAttr"

def hasTempAttrCaseSensitive = thetemp.hasAttribute("Temperature")
if (hasTempAttrCaseSensitive) {
    log.debug "${thetemp.displayName} supports the Temperature attribute."
} else {
    // this block will execute, since attribute names are case sensitive
    log.debug "${thetemp.displayName} does NOT support the Temperature attribute."
}

...

```

284.10.24 hasCapability()

Determine if this Device supports the specified capability name.

Tip: Capability names are case-sensitive.

Signature: Boolean hasCapability(String capabilityName)

Parameters: String capabilityName - the name of the capability to check if the Device supports.

Returns: Boolean - true if this Device has the specified capability. Returns a non-true value if not (may be null).

Example:

```

preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def hasSwitch = theswitch.hasCapability("Switch")
def hasSwitchCaseSensitive = theswitch.hasCapability("switch")
def hasPower = theswitch.hasCapability("Power")

// true
log.debug "${theswitch.displayName} has Switch capability? $hasSwitch"

if (!hasSwitchCaseSensitive) {
    // enters this block (names are case-sensitive!)
    log.debug "${theswitch.displayName} does not have the switch capability"
}

// true
log.debug "${theswitch.displayName} also has Power capability? $multiCapabilities"

...

```

284.10.25 hasCommand()

Determine if this Device has the specified command name.

Tip: Command names are case-sensitive.

Signature: Boolean hasCommand(String commandName)

Parameters: String commandName - the name of the command to check if the Device supports.

Returns: Boolean - true if this Device has the specified command. Returns a non-true value if not (may be null).

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "switchlevel", "capability.switchLevel"
    }
}
...

def hasOn = theswitch.hasCommand("on")
def hasOnCaseSensitive = theswitch.hasCommand("On")

// true
log.debug "${theswitch.displayName} has on command? $hasOn"

if (!hasOnCaseSensitive) {
    // enters this block - case-sensitive!
    log.debug "${theswitch.displayName} does not have On command"
}

def hasSetLevelCommand = switchlevel.hasCommand("setLevel")
// true
log.debug "${switchlevel.displayName} has command setLevel? $hasSetLevelCommand"
...
```

284.10.26 latestState()

Get the latest Device State record for the specified attribute.

Signature: State latestState(String attributeName)

Parameters: String attributeName - The name of the attribute to get the State record for.

Returns: State (page 1040) - The latest State record for the attribute specified for this Device.

Example:

```
def latestDeviceState = somedevice.latestState("someAttribute")
log.debug "latest state value: ${latestDeviceState.value}"
```

284.10.27 latestValue()

Get the latest reported value for the specified attribute.

Signature: `Object latestValue(String attributeName)`

Parameters: `String attributeName` - the name of the attribute to get the latest value for.

Returns: `Object` - the latest reported value. The exact type returned will vary depending upon the attribute.

Warning: The exact returned type for various attributes is not adequately documented at this time. Until they are, we recommend that you save often and experiment, or even look at the specific Device Handler for the device you are working with.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def switchattr = theswitch.latestValue("switch")
def tempattr = thetemp.latestValue("temperature")

log.debug "current switch: $switchattr"
log.debug "current temp: $tempattr"

// switch attribute returned as a string
log.debug "switchattr instanceof String? ${switchattr instanceof String}"

// temperature attribute returned as a Number
log.debug "tempattr instanceof Number? ${tempattr instanceof Number}"
...
```

284.10.28 statesBetween()

Get a list of Device *State* (page 1040) objects for the specified attribute between the specified times in reverse chronological order (newest first).

Note: Only State instances from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero State objects.

Signature: `List<State> statesBetween(String attributeName, Date startDate, Date endDate [, Map options])`

Parameters: `String attributeName` - The name of the attribute to get the States for.

`Date startDate` - The beginning date for the query.

`Date endDate` - The end date for the query.

`Map options` (*optional*) - options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return. By default, the maximum is 10.

Returns: `List <State (page 1040)>` - A list of State objects between the dates specified. A maximum of 1000 `State (page 1040)` objects will be returned.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def start = new Date() - 5
def end = new Date() - 1

def theStates = theswitch.statesBetween("switch", start, end)
log.debug "There are ${theStates.size()} between five days ago and yesterday"
...
```

284.10.29 statesSince()

Get a list of Device `State (page 1040)` objects for the specified attribute since the date specified.

Note: Only State instances from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero State objects.

Signature: `List<State> statesSince(String attributeName, Date startDate [, Map options])`

Parameters: `String attributeName` - The name of the attribute to get the States for.

`Date startDate` - The beginning date for the query.

`Map options (optional)` - options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return. By default, the maximum is 10.

Returns: `List <State (page 1040)>` - A list of State records since the specified start date. A maximum of 1000 `State (page 1040)` instances will be returned.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def theStates = theswitch.statesSince("switch", new Date() -3)
log.debug "There are ${theStates.size()} State records in the last 3 days"
...
```

284.11 Event

Events are core to the SmartThings platform. They allow SmartApps to respond to changes in the physical environment, and build automations around them.

Event instances are not created directly by SmartApp or Device Handlers. They are created internally by the SmartThings platform, and passed to SmartApp event handlers that have subscribed to those events.

Note: In a SmartApp or Device Handler, the method `createEvent()` exists to create a Map that defines properties of an Event. Only by returning the resulting map from a Device Handler's `parse()` method is an actual Event instance created and propagated through the SmartThings system.

The reference documentation here lists all methods available on an Event object instance.

284.11.1 `getData()`

A map of any additional data on the Event.

Signature: `String getData()`

Returns: `String` - A JSON string representing a map of the additional data (if any) on the Event.

Example:

Consider an Event created like this:

```
createEvent(name: "myevent", value: "myvalue", data: [key1: "val", key2: 42])
```

Then in an event handler method, we can get at the data like this:

```
def eventHandler(evt) {
    def data = parseJson(evt.data)
    log.debug "event data: ${data}"
    log.debug "event key1: ${data.key1}"
    log.debug "event key2: ${data.key2}"
}
```

284.11.2 `getDate()`

Acquisition time of this device state record.

Signature: `Date getDate()`

Returns: `Date` - the date and time this Event record was created.

Example:

```
def eventHandler(evt) {
    log.debug "event created at: ${evt.date}"
}
```

284.11.3 getDateValue()

The value of the Event as a `Date` object, if applicable.

Signature: `Date getDateValue()`

Returns: `Date` - If the value of this Event is date, a `Date` will be returned. `null` will be returned if the value of the Event is not parseable to a `Date`.

Example:

```
def eventHandler(evt) {
    // get the value of this event as a Date
    log.debug "The dateValue of this event is ${evt.dateValue}"
    log.debug "evt.dateValue instanceof Date? ${evt.dateValue instanceof Date}"
}
```

284.11.4 getDescription()

The raw description that generated this Event.

Signature: `String getDescription()`

Returns: `String` - the raw description that generated this Event.

Example:

```
def eventHandler(evt) {
    log.debug "event raw description: ${evt.description}"
}
```

284.11.5 getDescriptionText()

The description of the Event that is to be displayed to the user in the mobile application.

Signature: `String getDescriptionText()`

Returns: `String` - the description of this Event to be displayed to the user in the mobile application.

Example:

```
def eventHandler(evt) {
    log.debug "event description text: ${evt.descriptionText}"
}
```

284.11.6 getDevice()

The *Device* (page 1003) associated with this Event.

Signature: `Device getDevice()`

Returns: *Device* (page 1003) - the Device associated with this Event, or `null` if no Device is associated with this Event.

284.11.7 getDisplayName()

Signature: `String getDisplayName()`

Returns: `String` - The user-friendly name of the source of this Event. Typically the user-assigned device label.

Example:

```
def eventHandler(evt) {
    log.debug "event display name: ${evt.displayName}"
}
```

284.11.8 getDeviceId()

The unique system identifier of the *Device* (page 1003) associated with this Event.

Signature: `String getDeviceId()`

Returns: `String` - the unique system identifier of the device associated with this Event, or null if there is no device associated with this Event.

Example:

```
def eventHandler(evt) {
    log.debug "The device id for this event: ${evt.deviceId}"
}
```

284.11.9 getId()

The unique system identifier for this Event.

Signature: `String getId()`

Returns: `String` - the unique device identifier for this Event.

Example:

```
def eventHandler(evt) {
    log.debug "event id: ${evt.id}"
}
```

284.11.10 getDoubleValue()

The value of this Event, if the value can be parsed to a Double.

Signature: `Double getDoubleValue()`

Returns: `Double` - the value of this Event as a Double.

Warning: `doubleValue` will throw an Exception if the value of the Event is not parseable to a Double. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    // get the value of this event as an Double
    // throws an exception of the value is not convertable to a Double
    try {
        log.debug "The doubleValue of this event is ${evt.doubleValue}"
        log.debug "evt.doubleValue instanceof Double? ${evt.doubleValue instanceof Double}"
    } catch ( ) {
        log.debug("Trying to get the doubleValue for ${evt.name} threw an exception", e)
    }
}
```

284.11.11 getFloatValue()

The value of this Event as a Float, if it can be parsed into a Float.

Signature: Float getFloatValue()

Returns: Float - the value of this Event as a Float.

Warning: floatValue will throw an Exception if the Event's value is not parseable to a Float. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    // get the value of this event as an Float
    // throws an exception if not convertable to Float
    try {
        log.debug "The floatValue of this event is ${evt.floatValue}"
        log.debug "evt.floatValue instanceof Float? ${evt.floatValue instanceof Float}"
    } catch ( ) {
        log.debug("Trying to get the floatValue for ${evt.name} threw an exception", e)
    }
}
```

284.11.12 getHubId()

The unique system identifier of the Hub associated with this Event.

Signature: String getHubId()

Returns: String - the unique system identifier of the Hub associated with this Event, or null if no Hub is associated with this Event.

Example:

```
def eventHandler(evt) {
    log.debug "The hub id associated with this event: ${evt.hubId}"
}
```


284.11.13 getInstalledSmartAppId()

The unique system identifier of the SmartApp instance associated with this Event.

Signature: `String getInstalledSmartAppId()`

Returns: `String` - the unique system identifier of the SmartApp instance associated with this Event.

Example:

```
def eventHandler(evt) {
    log.debug "The installed SmartApp id associated with this event: ${evt.installedSmartAppId}"
}
```

284.11.14 getIntegerValue()

The value of this Event as an Integer.

Signature: `Integer getIntegerValue()`

Returns: `Integer` - the value of this Event as an Integer.

Warning: `integerValue` throws an Exception of the Event value cannot be parsed to an Integer. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    // get the value of this event as an Integer
    // throws an exception if not convertible to Integer
    try {
        log.debug "The integerValue of this event is ${evt.integerValue}"
        log.debug "The integerValue of this event is an Integer: ${evt.integerValue instanceof Integer}"
    } catch (e) {
        log.debug("Trying to get the integerValue for ${evt.name} threw an exception", e)
    }
}
```

284.11.15 getIsoDate()

Acquisition time of this Event as an ISO-8601 String.

Signature: `String getIsoDate()`

Returns: `String` - The acquisition time of this Event as an ISO-8601 String.

Example:

```
def eventHandler(evt) {
    log.debug "event isoDate: ${evt.isoDate}"
}
```

284.11.16 getJsonValue()

Value of the Event as a parsed JSON data structure.

Signature: Object getJsonValue()

Returns: Object - The value of the Event as a JSON structure

Warning: jsonValue throws an Exception if the value of the Event cannot be parsed into a JSON object. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    // get the value of this event as a JSON structure
    // throws an exception if the value is not convertible to JSON
    try {
        log.debug "The jsonValue of this event is ${evt.jsonValue}"
    } catch (e) {
        log.debug("Trying to get the jsonValue for ${evt.name} threw an exception", e)
    }
}
```

284.11.17 getLinkText()

Warning: Deprecated.
getLinkText() is deprecated. Use *getDisplayname()* (page 1019) instead.

The user-friendly name of the source of this Event. Typically the user-assigned device label.

284.11.18 getLocation()

The Location associated with this Event.

Signature: Location getLocation()

Returns: Location (page 1036) - The Location associated with this Event, or null if no Location is associated with this Event.

284.11.19 getLocationId()

The unique system identifier for the Location (page 1036) associated with this Event.

Signature: String getLocationId()

Returns: String - the unique system identifier for the Location (page 1036) associated with this Event.

284.11.20 getLongValue()

The value of this Event as a Long.

Signature: Long getLongValue()

Returns: Long - the value of this Event as a Long.

Warning: longValue throws an Exception if the value of the Event cannot be parsed to a Long. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    // get the value of this event as an Long
    // throws an exception if not convertible to Long
    try {
        def evtLongValue = evt.longVaue
        log.debug "The longValue of this event is evtLongValue"
        log.debug "evt.longValue instanceof Long? ${evtLongValue instanceof Long}"
    } catch ( ) {
        log.debug("Trying to get the longValue for ${evt.name} threw an exception", e)
    }
}
```

284.11.21 getName()

The name of this Event.

Signature: String getName()

Returns: String - the name of this Event.

Example:

```
def eventHandler(evt) {
    log.debug "the name of this event: ${evt.name}"
}
```

284.11.22 getNumberValue()

The value of this Event as a Number.

Signature: Number getNumberValue()

Returns: Number - the value of this Event as a Number.

Warning: numberValue throws an Exception if the value of the Event cannot be parsed to a Number. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    // get the value of this event as an Number
    // throws an exception if the value is not convertable to a Number
    try {
        def evtNumberValue = evt.numberValue
        log.debug "The numberValue of this event is ${evtNumberValue}"
        log.debug "evt.numberValue instanceof Number? ${evtNumberValue instanceof Number}"
    } catch ( ) {
        log.debug("Trying to get the numberValue for ${evt.name} threw an exception", e)
    }
}
```

284.11.23 getNumericValue()

The value of this Event as a Number.

Signature: Number getNumericValue()

Returns: Number - the value of this Event as a Number.

Warning: numericValue throws an Exception if the value of the Event cannot be parsed to a Number. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    // get the value of this event as an Number
    // throws an exception if the value is not convertable to a Number
    try {
        def evtNumberValue = evt.numericValue
        log.debug "The numericValue of this event is ${evtNumberValue}"
        log.debug "evt.numericValue instanceof Number? ${evtNumberValue instanceof Number}"
    } catch ( ) {
        log.debug("Trying to get the numericValue for ${evt.name} threw an exception", e)
    }
}
```

284.11.24 getSource()

The source of the Event.

Signature: String getSource()

Returns: String - the source of the Event. The following table lists the possible sources and their meaning:

Source	Description
"APP"	Event originated by an app touch Event in the mobile application.
"APP_COMMAND"	Event originated by using the mobile application (for example, using the mobile application to turn a light off)
"COMMAND"	Event originated by a SmartApp or Device Handler calling a command on a device.
"DEVICE"	Event originated by the physical actuation of a device.
"HUB"	Event originated on the Hub.
"LOCATION"	Event originated by a Location state change (for example, sunrise and sunset events)
"USER"	

Example:

```
def eventHandler(evt) {  
    log.debug "The source of this event is: ${evt.source}"  
}
```

284.11.25 getStringValue()

The value of this Event as a String.

Signature: String getStringValue()

Returns: String - the value of this Event as a String.

Example:

```
def eventHandler(evt) {  
    log.debug "The value of this event as a string: ${evt.stringValue}"  
}
```

284.11.26 getUnit()

The unit of measure for this Event, if applicable.

Signature: String getUnit()

Returns: String - the unit of measure of this Event, if applicable. null otherwise.

Example:

```
def eventHandler(evt) {  
    log.debug "The unit for this event: ${evt.unit}"  
}
```

284.11.27 getValue()

The value of this Event as a String.

Signature: String getValue()

Returns: String - the value of this Event as a String.

Example:

```
def eventHandler(evt) {  
    log.debug "The value of this event as a string: ${evt.value}"  
}
```

284.11.28 getXyzValue()

Value of the Event as a 3-entry Map with keys 'x', 'y', and 'z' with BigDecimal values. For example:

```
[x: 1001, y: -23, z: -1021]
```

Typically only useful for getting position data from the “Three Axis” Capability.

Signature: Map<String, BigDecimal> getXyzValue()

Returns: Map < String , BigDecimal > - A map representing the X, Y, and Z coordinates.

Warning: xyzValue throws an Exception if the value of the Event cannot be parsed to an X-Y-Z data structure. You should wrap calls in a try/catch block.

Example:

```
def positionChangeHandler(evt) {
    // get the value of this event as a 3 entry map with keys
    // 'x', 'y', 'z', and BigDecimal values
    // throws an exception if the value is not convertable to a Date
    try {
        log.debug "The xyzValue of this event is ${evt.xyzValue }"
        log.debug "evt.xyzValue instanceof Map? ${evt.xyzValue instanceof Map}"
    } catch ( ) {
        log.debug("Trying to get the xyzValue for ${evt.name} threw an exception", e)
    }
}
```

284.11.29 isDigital()

true if the Event is from the digital actuation (non-physical) of a Device, false otherwise.

Signature: Boolean physical()

Returns: Boolean - true if the Event is from the digital actuation of a Device, false otherwise.

Example:

```
def eventHandler(evt) {
    log.debug "event from digital actuation? ${evt.isDigital()}"
}
```

284.11.30 isPhysical()

true if the Event is from the physical actuation of a Device, false otherwise.

Signature: Boolean physical()

Returns: Boolean - true if the Event is from the physical actuation of a Device, false otherwise.

Example:

```
def eventHandler(evt) {
    log.debug "event from physical actuation? ${evt.isPhysical()}"
}
```

284.11.31 isStateChange()

true if the Attribute value for this Event is different than the previous one.

Signature: Boolean stateChange()

Returns: Boolean - true if the Attribute value for this Event is different than the previous one.

Example:

```
def eventHandler(evt) {
    log.debug "Is this event a state change? ${evt.isStateChange()}"
}
```

284.12 Hub

The Hub object encapsulates information about the Hub.

Here's a code snippet of a SmartApp that logs all available information for the Hub when the SmartApp is installed:

```
def installed() {
    def hub = location.hubs[0]

    log.debug "id: ${hub.id}"
    log.debug "zigbeeId: ${hub.zigbeeId}"
    log.debug "zigbeeEui: ${hub.zigbeeEui}"

    // PHYSICAL or VIRTUAL
    log.debug "type: ${hub.type}"

    log.debug "name: ${hub.name}"
    log.debug "firmwareVersionString: ${hub.firmwareVersionString}"
    log.debug "localIP: ${hub.localIP}"
    log.debug "localSrvPortTCP: ${hub.localSrvPortTCP}"
}
```

Below are the available properties on a Hub:

284.12.1 getFirmwareVersionString()

Signature: String getFirmwareVersionString() ()

Returns: String - The firmware version of the Hub

Example:

```
List getRealHubFirmwareVersions() {  
    return location.hubs*.firmwareVersionString.findAll { it }  
}
```

284.12.2 getId()

The unique system identifier for this Hub.

Signature: String getId()

Returns: String - the unique device identifier for this Hub.

Example:

```
def eventHandler(evt) {  
    log.debug "Hub ID associated with event: ${evt?.hub.id}"  
}
```

284.12.3 getLocalIP()

The local IP address of the Hub.

Signature: String getLocalIP()

Returns: String - The IP address of the Hub.

284.12.4 getLocalSrvPortTCP()

The local server TCP port of the Hub.

Signature: String getLocalSrvPortTCP()

Returns: String - the TCP port for the Hub.

284.12.5 getName()

The name of the Hub.

Signature: String getName()

Returns: String - the name of the Hub.

284.12.6 getType()

The type of the Hub. Either “PHYSICAL” or “VIRTUAL”.

Signature: `String getType()`

Returns: `String` - the type of the Hub.

284.12.7 getZigbeeEui()

The ZigBee Extended Unique Identifier of the Hub.

Signature: `String getZigbeeEui()`

Returns: `String` - The ZigBee EUI

284.12.8 getZigbeedId()

The ZigBee ID of the Hub.

Signature: `String getZigbeeId()`

Returns: `String` - the ZigBee ID

284.13 HubAction

The HubAction class is used to encapsulate a detailed request string while communicating with the devices in your network. For example, this request string can be used for search and discovery of devices in your network. HubAction can also be used in a SmartApp or in a Device Handler to communicate with the device.

Signature: `HubAction myhubAction = new physicalgraph.device.HubAction(String action, Protocol protocol)`

```
HubAction myhubAction = new physicalgraph.device.HubAction(String action,
Protocol protocol, String dni, Map options)
```

```
HubAction myhubAction = new physicalgraph.device.HubAction(Map params,
String dni = null, [Map options])
```

Parameters: `String action` - A string comprised of the request details targeted for the device.

`Protocol protocol` - Specific protocol to be used. Default value is `Protocol.LAN`.

`String dni` - Device Network ID of the device. Default value is null. For dni, we recommend using MAC address and not use IP and port numbers.

`Map options` - Default value is null. Available options are:

Option	Description
callback	Name of the callback method
type	Type of LAN request. Allowed values are: LAN_TYPE_CLIENT, LAN_TYPE_SERVER. Default value is LAN_TYPE_CLIENT.
protocol	Allowed values are LAN_PROTOCOL_TCP and LAN_PROTOCOL_UDP and default value is LAN_PROTOCOL_TCP. Note the difference in allowed values of this parameter when used in Maps params and Protocol protocol signatures.

Map params - Available parameters are:

Parameter	Description
path	Allowed values are any string of the form "/somepath". Default value is "/".
method	Allowed values are "POST", "GET", "PUT" and "PATCH". Default value is "POST".
protocol	Allowed values are Protocol.LAN. Default value is also Protocol.LAN.
headers	A map of HTTP headers. The HOST should be the "IP": "port" string of the device. Default values are ['Accept': '*/*', 'User-Agent': 'Linux UPnP/1.0 SmartThings',]. If 'Content-Type' is not included, then it is set to 'application/json' if params:body is a Map; otherwise 'Content-Type' is set to 'text/xml; charset="utf-8"'.
query	A map of URL query parameters.
body	Request body.

Example:

Send a device discovery command string to look for Samsung SmartCam device in your LAN network, via SSDP protocol.

```
// Send a device discovery command string to look for Samsung SmartCam device in your LAN network, via
sendHubCommand(new physicalgraph.device.HubAction("lan discovery urn:schemas-upnp-org:device:WANDevice:1"))
```

Note: Typically, the device discovery is done mainly via SSDP protocol. After the device is discovered, either REST or UPnP calls can be made for verification and communication with the device.

See *Building the Service Manager* (page 562) for more information.

Also note that, in the above example, while "urn:schemas-upnp-org:device:WANDevice:1" portion of the request string represents the notation defined by UPnP standard for device types, the terms "lan" and "discovery" are SmartThings-specific terms.

After the device is discovered, additional device information, such as device IP, MAC, port id, is available. Now it is possible to interact with the device using this additional information. Send a HubAction to the device as shown below, where myMAC is the MAC address string of the SmartCam device and calledBackHandler is the name of the method that is to be called when the device responds to this HubAction request object.

```
sendHubCommand(new physicalgraph.device.HubAction("GET /xml/device_description.xml HTTP/1.1\r\nHOST:myMAC\n\n"))
...
// the below calledBackHandler() is triggered when the device responds to the sendHubCommand() with '
void calledBackHandler(physicalgraph.device.HubResponse hubResponse) {
    log.debug "Entered calledBackHandler()..."
}
```

```

def body = hubResponse.xml
def devices = getDevices()
def device = devices.find { it?.key?.contains(body?.device?.UDN?.text()) }
if (device) {
    device.value << [name: body?.device?.roomName?.text(), model: body?.device?.modelName?.text()]
}
log.debug "device in calledBackHandler() is: ${device}"
log.debug "body in calledBackHandler() is: ${body}"
}

```

Returns: HubAction object.

Note: A Device Handler's `parse()` method can also return a HubAction object, in addition to the above-described usage by explicitly calling `sendHubCommand`.

See *Building the Device Type* (page 567) for more information.

284.14 InstalledSmartApp

The `InstalledSmartApp` class represents a `SmartApp`. Every `SmartApp` has an instance of `InstalledSmartApp` available to it via the `app` object. `InstalledSmartApp` is also used in *Parent-Child SmartApps* (page 393), specifically it is the return type of `addChildApp()` (page 900).

284.14.1 currentState()

Gets the current state of the given attribute.

Signature: `AppState currentState(String attributeName)`

Parameters: `String attributeName` - The attribute name to get the state for

Returns: `AppState` (page 981) - The latest state information for the specified attribute

Example:

```

...
def myAppState = app.currentState("someAttribute")
log.debug "state value: ${myAppState.value}"
...

```

284.14.2 getAccountId()

Gets the account ID of the owner of the Location in to which this `SmartApp` is installed.

Signature: `String getAccountId()`

Returns: `String` - The account ID of the owner of the Location in to which this `SmartApp` is installed.

Example:

```
def accountId = app.getAccountId()
log.debug "This account ID of this installed SmartApp is $accountId"
```

284.14.3 getAllChildApps()

Gets a list of child SmartApps associated with this SmartApp. This returns child SmartApps that have both “complete” and “incomplete” *installation states* (page 1033).

Signature: `def getAllChildApps()`

Returns: `List < InstalledSmartApp` (page 1031) `>` - A list of child SmartApps

Example:

```
def childApps = app.getAllChildApps()
log.debug "The app has ${childApps.size()} child SmartApps"
```

284.14.4 getAppSettings()

Gets the settings currently associated with this SmartApp.

Note: This method applies to the SmartApp’s *Private settings* (page 312).

Signature: `Map app.getAppSettings()`

Returns: `Map` - A map of key, value pairs that represent the current SmartApp settings

284.14.5 getChildApps()

Gets a list of child apps associated with this SmartApp. This only returns child SmartApps that have an *installation states* (page 1033) of “complete”.

Signature: `def getChildApps()`

Returns: `List < InstalledSmartApp` (page 1031) `>` - A list of child SmartApps

Example:

```
def childApps = app.childApps

// Update the label for all child apps
childApps.each {
    if (!it.label?.startsWith(app.name)) {
        it.updateLabel("$app.name/${it.label}")
    }
}
```

284.14.6 getChildDevices()

Gets a list of child devices associated with this SmartApp.

Signature: `List<Device> getChildDevices()`

Returns: `List < Device` (page 1003) `>` - A list of child devices

Example:

```
def children = app.getChildDevices()
log.debug "SmartApp with id $app.id has ${children.size()} child devices"

children.each { child ->
    log.debug "child device id $child.id with label $child.label"
}
```

284.14.7 getExecutionIsModeRestricted()

Returns `true` if the SmartApp's execution is restricted by modes. The restrictive modes would have been configured when the SmartApp was installed.

Signature: `Boolean getExecutionIsModeRestricted()`

Returns: `Boolean` - True if the execution of the SmartApp is restricted to certain modes

284.14.8 getExecutableModes()

Get a list of modes that this SmartApp is allowed to execute in.

Signature: `Mode` (page 1039) `getExecutableModes()`

Returns: `Mode` (page 1039) - A list of modes that this SmartApp is allowed to execute in

284.14.9 getId()

Get the id of the SmartApp

Signature: `String getId()`

Returns: The ID of the SmartApp

284.14.10 getInstallationState()

Get the current installation state of the SmartApp.

Signature: `String getInstallationState()`

Returns: The current installation state of the SmartApp. Can be `incomplete` or `complete`

284.14.11 getLabel()

Get the label of the SmartApp

Signature: `String getLabel()`

Returns: The label of the SmartApp

284.14.12 getName()

Get the name of the SmartApp

Signature: `String getName()`

Returns: The name of the SmartApp

284.14.13 getNamespace()

Get the namespace of the SmartApp

Signature: `String getNamespace()`

Returns: The namespace of the SmartApp

284.14.14 getParent()

Gets the parent of the SmartApp.

Signature: `InstalledSmartApp (page 1031) getParent()`

Returns: `InstalledSmartApp (page 1031)` - The parent of this SmartApp

284.14.15 getSubscriptions()

Signature: `List<EventSubscriptionWrapper> getSubscriptions()`

Returns `List<EventSubscriptionWrapper[]` - A list of subscriptions associated with this SmartApp

284.14.16 statesBetween()

Get a list of app *AppState* (page 981) objects for the specified attribute between the specified times in reverse chronological order (newest first).

Note: Only State instances from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero State objects.

Signature: `List<AppState> statesBetween(String attributeName, Date startDate, Date endDate [, Map options])`

Parameters: `String attributeName` - The name of the attribute to get the States for.

`Date startDate` - The beginning date for the query.

`Date endDate` - The end date for the query.

`Map options` (*optional*) - options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return. By default, the maximum is 10.

Returns: `List<AppState>` (page 981) - A list of State objects between the dates specified. A maximum of 1000 *State* (page 1040) objects will be returned.

Example:

```

...
def start = new Date() - 5
def end = new Date() - 1

def theStates = app.statesBetween("myAttribute", start, end)
log.debug "There are ${theStates.size()} between five days ago and yesterday"
...

```

284.14.17 statesSince()

Get a list of app *AppState* (page 981) objects for the specified attribute since the date specified.

Note: Only State instances from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero State objects.

Signature: `List<AppState> statesSince(String attributeName, Date startDate [, Map options])`

Parameters: `String attributeName` - The name of the attribute to get the States for.

`Date startDate` - The beginning date for the query.

`Map options` (*optional*) - options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return. By default, the maximum is 10.

Returns: `List<AppState>` (page 981) - A list of State records since the specified start date. A maximum of 1000 *State* (page 1040) instances will be returned.

Example:

```
def theStates = app.statesSince("myAttribute", new Date() -3)
log.debug "There are ${theStates.size()} State records in the last 3 days"
...
```

284.14.18 updateLabel()

Update the label of this SmartApp.

Signature: void updateLabel(String label)

Parameters: String label - The updated label value

Returns: void

284.15 Location

A Location represents a user's geo-location, such as "Home" or "office". Locations do not have to have a SmartThings Hub, but generally do.

All SmartApps and Device Handlers are injected with a `location` property that is the Location into which the SmartApp is installed.

284.15.1 getContactBookEnabled()

true if this Location has Contact Book enabled (has Contacts), false otherwise.

Signature: Boolean getContactBookEnabled()

Returns: true if this Location has Contact Book enabled (has Contacts), false otherwise.

284.15.2 getCurrentMode()

The current Mode for the Location.

Signature: Mode getCurrentMode()

Returns: Mode (page 1039) - The current mode for the Location.

Example:

```
log.debug "location.currentMode: ${location.currentMode}"
```

284.15.3 getId()

The unique internal system identifier for the Location.

Signature: `String getId()`

Returns: `String` - the unique internal system identifier for the Location.

Example:

```
log.debug "location.id: ${location.id}"
```

284.15.4 getHubs()

The list of Hubs for this Location. Currently only Hub can be installed into a Location, though this API returns a List to allow for future expandability.

Signature: `List<Hub> getHubs()`

Returns: `List<Hub>` (page 1027) - the Hubs for this Location.

Example:

```
log.debug "Hubs: ${location.hubs*.id}"
```

284.15.5 getLatitude()

Geographical latitude of the Location. Southern latitudes are negative. Requires that location services are enabled in the mobile app.

Signature: `BigDecimal getLatitude()`

Returns: `BigDecimal` - the latitude for the Location.

Example:

```
log.debug "location.latitude: ${location.latitude}"
```

284.15.6 getLongitude()

Geographical longitude of the Location. Western longitudes are negative. Requires that location services are enabled in the mobile app.

Signature: `BigDecimal getLongitude()`

Returns: `BigDecimal` - the longitude for the Location.

Example:

```
log.debug "location.longitude: ${location.longitude}"
```

284.15.7 getMode()

The current Mode name for the Location.

Signature: `String getMode()`

Returns: `String` - the name of the current Mode for the Location.

Example:

```
log.debug "location mode name: ${location.mode}"
```

284.15.8 getModes()

List of Modes for the Location.

Signature: `List<Mode> getModes()`

Returns: `List<Mode>` (page 1039) - the List of Modes for the Location.

Example:

```
log.debug "Modes for this Location: ${location.modes}"
```

284.15.9 getName()

The name of the Location, as assigned by the user.

Signature: `String getName()`

Returns: `String` - the name of the Location as assigned by the user.

Example:

```
log.debug "The name of this Location is: ${location.name}"
```

284.15.10 setMode()

Set the mode for this Location.

Signature: `void setMode(String mode) void setMode(Mode mode)`

Returns: `void`

Warning: `setMode()` will raise an error if the specified mode does not exist for the Location. You should verify the mode exists as in the example below.

Example:

```
def modeToSetTo = "Home"
if (location.modes?.find {it.name == modeToSetTo}) {
    location.setMode("Home")
}
```

284.15.11 getTemperatureScale()

The temperature scale (“F” for fahrenheit, “C” for celsius) for this Location.

Signature: `String getTemperatureScale()`

Returns: `String` - the temperature scale set for this Location. Either “F” for fahrenheit or “C” for celsius.

Example:

```
def tempscale = location.temperatureScale
log.debug "Temperature scale for this Location is $tempscale"
```

284.15.12 getTimeZone()

The time zone for the Location. Requires that location services are enabled in the mobile application.

Signature: `TimeZone getTimeZone()`

Returns: `TimeZone` - the time zone for the Location.

Example:

```
log.debug "The time zone for this Location is: ${location.timeZone}"
```

284.15.13 getZipCode()

The ZIP code for the Location, if in the USA. Requires that location services be enabled in the mobile application.

Signature: `String getZipCode()`

Returns: `String` - the ZIP code for the Location.

Example:

```
log.debug "The zip code for this Location: ${location.zipCode}"
```

284.16 Mode

Modes can be thought of as behavior filters for your home. Users want to change how things act or behave in their home based on the Mode you’re in.

SmartThings developers cannot create a new Mode. The most common way to interact with a Mode instance is by using the [Location](#) (page 1036) to get Mode information:

```
// Get the current Mode
def curMode = location.currentMode

// Get a list of all Modes for this location
def allModesForLocation = location.modes
```

284.16.1 getId()

The unique internal system identifier of the Mode.

Signature: String getId()

Returns: String - the unique internal system identifier for the Mode.

```
def curMode = location.currentMode
log.debug "The current Mode ID is: ${curMode.id}"
```

284.16.2 getName()

The name of the Mode.

Signature: String getName()

Returns: String - the name of the Mode, usually assigned by the user.

Example:

```
def curMode = location.currentMode
log.debug "The current mode name is: ${curMode.name}"
```

284.17 State

A State object encapsulates information about a particular *Attribute* (page 997) at a particular moment in time.

State objects are associated with a *Device* (page 1003) - a Device may have zero-to-many *Attribute* (page 997) s, and an Attribute has zero-to-many associated State records.

Refer to the Devices section of the SmartApp Guide for more information about the relationship between Devices, Attributes, and State.

A few ways to get a State object instance from a device (See the *Device* (page 1003) API reference for detailed information):

```
preferences {
    section() {
        input "thecontact", "capability.contactSensor"
    }
}
...
// <device>.<attributeName>State
def latestState = thecontact.contactState

// <device>.currentState(<attributeName>)
def latestState2 = thecontact.currentState("contact")

// get a list of states between two dates
def recentStates = thecontact.statesBetween(new Date() - 5, new Date())
```

284.17.1 getDate()

The date and time the State object was created.

Signature: Date getDate()

Returns: Date - the Date this State object was created.

Example:

```
def stateDate = contactSensor?.currentState("contact").date
```

284.17.2 getDateValue()

The value of the underlying attribute as a Date.

Signature: Date getDateValue()

Returns: Date - the value if the underlying attribute as a Date. Returns null if the attribute value cannot be parsed into a Date.

284.17.3 getDoubleValue()

The value of the underlying Attribute as a Double.

Signature: Double getDoubleValue()

Returns: Double - the value of the underlying attribute as a Double.

Warning: getDoubleValue() throws an Exception if the underlying attribute value cannot be parsed into a Double.
You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsDouble = someDevice.currentState("someAttribute").doubleValue
    log.debug "latestStateAsDouble: $latestStateAsDouble"
} catch (e) {
    log.debug "caught exception trying to get double for state record"
}
```

284.17.4 getFloatValue()

The value of the underlying Attribute as a Float.

Signature: Float getFloatValue()

Returns: Float - the value of the underlying Attribute as a Float.

Warning: `getFloatValue()` throws an Exception if the underlying attribute value cannot be parsed into a Double.
You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsFloat = someDevice.currentState("someAttribute").floatValue
    log.debug "latestStateAsFloat: $latestStateAsFloat"
} catch (e) {
    log.debug "caught exception trying to get floatValue for state record"
}
```

284.17.5 getId()

The unique system identifier for the State object.

Signature: `String getId()`

Returns: `String` - the unique system identifier for the State object.

Example:

```
def latestState = someDevice.currentState("someAttribute")
log.debug "latest state id: ${latestState.id}"
```

284.17.6 getIntegerValue()

The value of the underlying Attribute as an Integer.

Signature: `Integer getIntegerValue()`

Returns: `Integer` - the value of the underlying Attribute as a Integer.

Warning: `getIntegerValue()` throws an Exception if the underlying attribute value cannot be parsed into a Integer.
You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsInt = someDevice.currentState("someAttribute").integerValue
    log.debug "latestStateAsInt: $latestStateAsInt"
} catch (e) {
    log.debug "caught exception trying to get integerValue for state record"
}
```

284.17.7 getIsoDate()

The acquisition time of this State object as an ISO-8601 String

Signature: `String getIsoDate()`

Returns: `String` - the time this Sate object was created as an ISO-8601 Strring

Example:

```
def latestState = someDevice.currentState("someAttribute")
log.debug "latest state isoDate: ${latestState.isoDate}"
```

284.17.8 getJsonValue()

Value of the underlying Attribute parsed into a JSON data structure.

Signature: `Object getJsonValue()`

Returns: `Object` - the value if the underlying Attribute parsed into a JSON data structure.

Warning: `getJsonValue()` throws an Exception of the underlying attribute value cannot be parsed into a Integer.
You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsJsonValue = someDevice.currentState("someAttribute").jsonValue
    log.debug "latestStateAsJsonValue: $latestStateAsJsonValue"
} catch (e) {
    log.debug "caught exception trying to get jsonValue for state record"
}
```

284.17.9 getLongValue()

The value of the underlying Attribute as a Long.

Signature: `Long getLongValue()`

Returns: `Long` - the value if the underlying Attribute as a Long.

Warning: `getLongValue()` throws an Exception of the underlying attribute value cannot be parsed into a Long.
You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsLong = someDevice.currentState("someAttribute").longValue
    log.debug "latestStateAsLong: $latestStateAsLong"
} catch (e) {
    log.debug "caught exception trying to get longValue for state record"
}
```

284.17.10 getName()

The name of the underlying Attribute.

Signature: String getName()

Returns: String - the name of the underlying Attribute.

Example:

```
def latest = contactSensor.currentState("contact")
log.debug "name: ${latest.name}"
```

284.17.11 getNumberValue()

The value of the underlying Attribute as a BigDecimal.

Signature: BigDecimal getNumberValue()

Returns: Number - the value if the underlying Attribute as a Number.

Warning: getNumberValue() throws an Exception of the underlying attribute value cannot be parsed into a getNumberValue().
You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsNumber = someDevice.currentState("someAttribute").numberValue
    log.debug "latestStateAsNumber: $latestStateAsNumber"
} catch (e) {
    log.debug "caught exception trying to get numberValue for state record"
}
```

284.17.12 getNumericValue()

The value of the underlying Attribute as a Number.

Signature: Number getNumericValue()

Returns: Number - the value if the underlying Attribute as a Number.

Warning: getNumericValue() throws an Exception of the underlying attribute value cannot be parsed into a Number.
You should wrap calls in a try/catch block.

Example:


```
try {
    def latestStateAsNumber = someDevice.currentState("someAttribute").numericValue
    log.debug "latestStateAsNumber: $latestStateAsNumber"
} catch (e) {
    log.debug "caught exception trying to get numericValue for state record"
}
```

284.17.13 getStringValue()

The value of the underlying Attribute as a String

Signature: String getStringValue()

Returns: String - the value of the underlying Attribute as a String.

Example:

```
def latest = contactSensor.currentState("contact")
log.debug "stringValue: ${latest.stringValue}"
```

284.17.14 getUnit()

The unit of measure for the underlying Attribute.

Signature: String getUnit()

Returns: String - the unit of measure for the underlying Attribute, if applicable, null otherwise.

Example:

```
def latest = tempSensor.currentState("temperature")
log.debug "unit: ${latest.unit}"
```

284.17.15 getValue()

The value of the underlying Attribute as a String

Signature: String getUnit()

Returns: String - the value of the underlying Attribute as a String.

Example:

```
def latest = contactSensor.currentState("contact")
log.debug "stringValue: ${latest.value}"
```

284.17.16 getXyzValue()

Value of the underlying Attribute as a 3-entry Map with keys 'x', 'y', and 'z' with BigDecimal values. For example:

```
[x: 1001, y: -23, z: -1021]
```

Typically only useful for getting position data from the “Three Axis” Capability.

Signature: `Map<String, BigDecimal> getXyzValue()`

Returns: `Map < String , BigDecimal >` - A map representing the X, Y, and Z coordinates.

Warning: `xyzValue` throws an Exception if the value of the Event cannot be parsed to an X-Y-Z data structure. You should wrap calls in a try/catch block.

Example:

```
def latest = threeAxisDevice.currentState("threeAxis")

// get the value of this event as a 3 entry map with keys
// 'x', 'y', 'z', and BigDecimal values
// throws an exception if the value is not convertible to a Date
try {
    log.debug "The xyzValue of this event is ${latest.xyzValue}"
    log.debug "latest.xyzValue instanceof Map? ${latest.xyzValue instanceof Map}"
} catch (e) {
    log.debug "Trying to get the xyzValue threw an exception: $e"
}
```

284.18 ZigBee Reference

Note: As of now, any time the ZigBee library logs a message, an error will be seen. This is a known issue and will be fixed with the next deploy.

The ZigBee library contains many shorthands and conveniences for developing ZigBee Device Handlers.

ZigBee devices have fingerprints that define what the device is when it joins a ZigBee network. Currently you define the expected fingerprint for a device in the Device Handler metadata block as part of the definition. An example would look like this:

```
metadata {
    // Automatically generated. Make future change here.
    definition (name: "SmartPower Outlet", namespace: "smarththings", author: "SmartThings") {
        capability "Actuator"
        capability "Switch"
        capability "Power Meter"
        capability "Configuration"
        capability "Refresh"
        capability "Sensor"

        // indicates that device keeps track of heartbeat (in state.heartbeat)
        attribute "heartbeat", "string"

        fingerprint profileId: "0104", inClusters: "0000,0003,0004,0005,0006,0B04,0B05", outClusters: "0000,0003,0004,0005,0006,0B04,0B05"
        fingerprint profileId: "0104", inClusters: "0000,0003,0004,0005,0006,0B04,0B05", outClusters: "0000,0003,0004,0005,0006,0B04,0B05"
        fingerprint profileId: "0104", inClusters: "0000,0003,0004,0005,0006,0B04,0B05", outClusters: "0000,0003,0004,0005,0006,0B04,0B05"
        fingerprint profileId: "0104", inClusters: "0000,0003,0004,0005,0006,0B04,0B05", outClusters: "0000,0003,0004,0005,0006,0B04,0B05"
    }
}
```

If the fingerprint declaration contains a value for both the `manufacturer` and `model` attributes, you have the option to add the `deviceJoinName` attribute.

Fingerprint Attribute	Type	Value
<code>deviceJoinName</code>	String	Overrides the device type name when pairing. This allows the developer to customize the device name while joining a ZigBee device.

284.18.1 Parse methods

`zigbee.getEvent()`

The `getEvent()` method will try to parse ZigBee Clusters into a map whose key/value pairs can be directly handled by the `sendEvent()` method.

Signature:

```
Map<String:String> zigbee.getEvent(String description)
```

Parameters:

- **description:** The description value passed to the parse method from the device.

Return Values: *Map: [name: <event name>, value: <event value>]*

Example

```
def parse(String description) {
    def result = zigbee.getEvent(description)

    if(result) {
        sendEvent(result)
    } else {
        // zigbee.getEvent was unable to parse description. Description must be parsed manually.
    }
}
```

The `zigbee.getEvent()` method can parse the following event types with work being done to for additional event types:

Event Type	Cluster value
switch	0x0006
level	0x0008
power	0x0702 and 0x0B04
colorTemperature	0x0300

Note: Only color temperature can be parsed. Full color control is in the works.

Note: For the power event type, the value can be reported in mW, W, or kW. This means that it is up to the developer to make adjustments to the value before calling `sendEvent` so it will be displayed correctly.

284.18.2 Low level commands

additionalParams

There are several ZigBee methods that support an optional map parameter called `additionalParams`. This is intended to be used to support future params without affecting backward compatibility. The following keys are supported:

Key	Type	Description
<code>mfgCode</code>	integer	The ZigBee manufacturing code (e.g. 0x110A)
<code>destEndpoint</code>	integer	The destination endpoint for the given message (e.g. 0x02)

zigbee.command()

Send a Cluster specific Command. This method is overloaded and has a couple of signatures.

Signature:

```
zigbee.command(Integer cluster, Integer command, [String... payload])
```

Parameters:

- **Cluster:** The Cluster ID
- **Command:** The Command ID
- **payload** (optional): Zero or more arguments required by the Command. Each argument should be passed as an ASCII hex string in little endian format of the appropriate width for the data type. For example, to pass the value 5 for a UINT24 (24-bit unsigned integer) you would pass "050000".

Signature:

```
zigbee.command(Integer cluster, Integer command, String payload, additionalParams = [:])
```

Parameters:

- **Cluster:** The Cluster ID
- **Command:** The Command ID
- **payload:** An ASCII hex string in little endian format of the appropriate width for the data type. For example, to pass the value 5 for a UINT24 (24-bit unsigned integer) you would pass "050000". You can also use the `DataType.pack()` (page 1058) method described below. If you have multiple arguments, they can just be appended in order.
- **additionalParams:** An optional map to specify additional parameters. See [additionalParams](#) (page 1048) for supported attributes.

Examples:

- **Send Move To Level Command to Level Control Cluster.**

```
zigbee.command(0x0008, 0x04, "FE", "0500")
```

Where *Level* equals 0xFE (full on) and *Transition Time* equals 0x0005 (5 seconds)

- **Send Off Command to On/Off Cluster.**

```
zigbee.command(0x0006, 0x00)
```

zigbee.readAttribute()

Read the current attribute value of the specified Cluster.

Signature:

```
zigbee.readAttribute(Integer cluster, Integer attributeId, Map additionalParams=[:])
```

Parameters:

- **Cluster:** The Cluster ID to read from
- **attributeId:** The ID of the attribute to read
- **additionalParams:** An optional map to specify additional parameters. See [additionalParams](#) (page 1048) for supported attributes.

Example:

- **Read *CurrentLevel* attribute of the *Level Control* Cluster.**

```
zigbee.readAttribute(0x0008, 0x0000)
```

- **Read a manufacturer specific attribute on the SmartThings multi-sensor**

```
zigbee.readAttribute(0xFC02, 0x0010, [mfgCode: 0x110A])
```

zigbee.writeAttribute()

Write the attribute value of the specified Cluster.

Signature:

```
zigbee.writeAttribute(Integer cluster, Integer attributeId, Integer dataType, value, Map additionalParams=[:])
```

Parameters:

- **Cluster:** The Cluster ID to write
- **attributeId:** The attribute ID to write
- **dataType:** The data type ID of the attribute as specified in the [ZigBee Cluster library](#)
- **value:** The Integer value to write for data types of *boolean*, *unsigned int*, *signed int*, *general data*, and *enumerations*. Other data types are not currently supported but will be added in the future. Let us know if you need a data type that is not currently supported.
- **additionalParams:** An optional map to specify additional parameters. See [additionalParams](#) (page 1048) for supported attributes.

Example:

- **Write the value 0x12AB to a unsigned 16-bit integer attribute**

```
zigbee.writeAttribute(0x0008, 0x0010, 0x21, 0x12AB)
```

- **Write a manufacturer specific attribute on the SmartThings multi-sensor**

```
zigbee.writeAttribute(0xFC02, 0x0000, 0x20, 1, [mfgCode: 0x110A])
```

zigbee.configureReporting()

Configure a ZigBee device's reporting properties. Refer to the *Configure Reporting* Command in the [ZigBee Cluster Library](#) for more information.

Signature:

```
zigbee.configureReporting(Integer cluster,  
    Integer attributeId, Integer dataType,  
    Integer minReportTime, Integer MaxReportTime,  
    [Integer reportableChange],  
    Map additionalParams={:})
```

Parameters:

- **Cluster:** The Cluster ID of the requested report
- **attributeId:** The attribute ID for the requested report
- **dataType:** The two byte ZigBee type value for the requested report (see *DataType* (page 1056))
- **minReportTime:** Minimum number of seconds between reports
- **maxReportTime:** Maximum number of seconds between reports
- **reportableChange** (optional): Amount of change needed to trigger a report. Required for analog data types. Discrete data types should always provide *null* for this value.
- **additionalParams:** An optional map to specify additional parameters. See *additionalParams* (page 1048) for supported attributes.

Examples:

- **Discrete data type**

```
zigbee.configureReporting(0x0006, 0x0000, 0x10, 0, 600, null)
```

- **Analog data type**

```
zigbee.configureReporting(0x0008, 0x0000, 0x20, 1, 3600, 0x01)
```

- **Configure a manufacturer specific report on the SmartThings multi-sensor**

```
zigbee.configureReporting(0xFC02, 0x0010, 0x18, 10, 3600, 0x01, [mfgCode: 0x110A])
```

284.18.3 ZigBee Capabilities

The following table outlines the Commands necessary to both configure and get updated information from ZigBee devices that support the capabilities outlined below.

Note: All methods outlined in the table need the `zigbee.` prefix

Capability	Configure	Refresh	Notes
Battery	configureReporting(0x0001, 0x0020, 0x20, 30, 21600, 0x01)		
Color Temp	configureReporting(0x0300, 0x0007, 0x21, 1, 3600, 0x10)	readAttribute(0x0300, 0x0007)	For devices that support the Color Control Cluster (0x0300)
Level	configureReporting(0x0008, 0x0000, 0x20, 1, 3600, 0x01)	readAttribute(0x0008, 0x0000)	
Power	configureReporting(0x0702, 0x0400, 0x2A, 1, 600, 0x05)	readAttribute(0x0704, 0x0400)	For devices that support the Metering Cluster (0x0704)
Power	configureReporting(0x0B04, 0x050B, 0x29, 1, 600, 0x0005)	readAttribute(0x0B04, 0x050B)	For devices that support the Electrical Measurement Cluster (0x0B04)
Switch	configureReporting(0x0006, 0x0000, 0x10, 0, 600, null)	readAttribute(0x0006, 0x0000)	
Temperature	configureReporting(0x0402, 0x0000, 0x29, 30, 3600, 0x0064)		

Examples:

- Get the latest *level* value from a dimmer switch. From the table above, we find the *level* capability and look at the **Refresh** column to find the correct Command to execute.

```
readAttribute(0x0008, 0x0000)
```

- Configure the *level* capability for a dimmer type switch. The configure reporting Command from the table above for *level* configures the device for a min reporting interval of 5 seconds, a reporting interval of 1 hour (3600 s) if there has been no activity, and a min level change of 01.

```
configureReporting(0x0008, 0x0000, 0x20, 1, 3600, 0x01)
```

The following utility methods are available as capability based Commands.

zigbee.on()

Sends the *on* Command, 0x01, to the *on/off* Cluster, 0x0006

Signature:

```
zigbee.on()
```

zigbee.off()

Sends the *off* Command, 0x00, to the *on/off* Cluster, 0x0006

Signature:

```
zigbee.off()
```

zigbee.setLevel()

Sends the *level* Command, 0x04, to the level control Cluster, 0x0008 with the passed in rate.

Signature:

```
zigbee.setLevel(Integer level, Integer rate)
```

Parameters:

- **level:** A value between 0-100 inclusive.
- **rate:** Time in tenths of a second. E.g. rate = 100 //max value translates to 10 seconds.

zigbee.setColorTemperature()

Sets the color to the specified temperature value in K.

Signature:

```
zigbee.setColorTemperature(Integer value)
```

Parameters:

- **value:** The temperature value to set the color to in K. Usually greater than 2700
-

284.18.4 ZigBee helper commands

zigbee.parseDescriptionAsMap()

Parses a device description into a map that contains data and capabilities.

Signature:

```
zigbee.parseDescriptionAsMap(String description)
```

Parameters:

- **description:** The description string from the device

zigbee.convertToHexString()

Convert the given value to a hex string of given width

Signature:

```
zigbee.convertToHexString(Integer value, Integer width)
```

Parameters:

- **value:** Integer value to be converted
- **width:** the minimum width of the hex string. Default value is 2

Examples:

```
zigbee.convertToHexString(10, 2) //result: 0A  
zigbee.convertToHexString(10, 4) //result: 000A
```


zigbee.convertHexToInt()

Convert the given hex value to an Integer

Signature:

```
zigbee.convertHexToInt (String value)
```

Parameters:

- **value:** The hex value to be converted to an Integer

Example:

```
zigbee.convertHexToInt("0001") //result: 1
zigbee.convertHexToInt("000F") //result: 15
def myInt = zigbee.convertHexToInt("12AB") // result = 4779. The result of calling Integer.parseInt()
    assert myInt == 4779 //true
    assert myInt == 0x12AB //also true
```

zigbee.hexNotEqual()

Returns true if the compared hex values are not equal.

Signature:

```
zigbee.hexNotEqual (String hex1, String hex2)
```

Parameters:

- **hex1:** Hex value to compare
- **hex2:** Hex value to compare against first value

zigbee.parseZoneStatus()

Returns a ZoneStatus object (see below) with the parsed value from the message description. The description should be of the form "zone status {number}" where {number} is a hex number.

Signature:

```
zigbee.parseZoneStatus (String description)
```

Parameters:

- **description:** A zone status message description.

284.18.5 Additional ZigBee classes

There are some additional classes that are provided to make interacting with and handling Zigbee messages easier.

ZoneStatus

The purpose of the ZoneStatus class is to handle the ZoneStatus attribute in the IAS Zone cluster. It has a single constructor that takes an int which is the ZoneStatus attribute value.

Constructor:

```
ZoneStatus(int zonestatus)
```

Example:

```
ZoneStatus zs = ZoneStatus(0x41) // Trouble & Alarm1
```

Accessing a Property/attribute

Once you have created the ZoneStatus object you can query the individual bits in a number of ways. First you can get the value of each individual bit (1 or 0) by accessing the property with the same name. The properties are as follows:

Bit Number	Property Name	Values
0	alarm1	1 - opened or alarmed 0 - closed or not alarmed
1	alarm2	1 - opened or alarmed 0 - closed or not alarmed
2	tamper	1 - tampered 0 - not tampered
3	battery	1 - low battery 0 - battery OK
4	supervisionReports	1 - reports 0 - does not report
5	restoreReports	1 - reports restore 0 - does not report restore
6	trouble	1 - trouble/failure 0 - OK
7	ac	1 - ac/mains fault 0 - ac/mains OK
8	test	1 - sensor is in test mode 0 - sensor is in operation mode
9	batteryDefect	1 - sensor detects a defective battery 0 - sensor battery is functioning normally

See the [ZigBee Home Automation \(HA\)](#) specification and the [ZigBee Cluster Library \(ZCL\)](#) specification for more information.

Example:

```
ZoneStatus zs = ZoneStatus(0x41) // Trouble & Alarm1
zs.alarm1 // 1
zs.alarm2 // 0
zs.trouble // 1
```

The ZoneStatus object also exposes a number of query methods for getting a true/false for each attribute value. They are as follows:

Property Name	Query method
alarm1	isAlarm1Set()
alarm2	isAlarm2Set()
tamper	isTamperSet()
battery	isBatterySet()
supervisionReports	isSupervisionReportsSet()
restoreReports	isRestoreReportsSet()
trouble	isTroubleSet()
ac	isAcSet()
test	isTestSet()
batteryDefect	isBatteryDefectSet()

Example of DTH parseIasMessage for a motion sensor:

```
private Map parseIasMessage(String description) {
    ZoneStatus zs = zigbee.parseZoneStatus(description)
    Map resultMap = [:]

    resultMap.name = 'motion'
    resultMap.value = zs.isAlarm1Set() ? 'active' : 'inactive'

    return resultMap
}
```

Data Type

The DataType class contains information and some utility methods for ZCL data types.

DataType constants

The list of types and their DataType constant name are as follows:

ZCL Data Type	DataType constant name	ZCL numeric value
No Data	NO_DATA	0x00
8-bit data	DATA8	0x08
16-bit data	DATA16	0x09
24-bit data	DATA24	0x0a
32-bit data	DATA32	0x0b
40-bit data	DATA40	0x0c
48-bit data	DATA48	0x0d
56-bit data	DATA56	0x0e
64-bit data	DATA64	0x0f
Boolean	BOOLEAN	0x10
8-bit bitmap	BITMAP8	0x18

Continued on next page

Table 284.1 – continued from previous page

ZCL Data Type	Data Type constant name	ZCL numeric value
16-bit bitmap	BITMAP16	0x19
24-bit bitmap	BITMAP24	0x1a
32-bit bitmap	BITMAP32	0x1b
40-bit bitmap	BITMAP40	0x1c
48-bit bitmap	BITMAP48	0x1d
56-bit bitmap	BITMAP56	0x1e
64-bit bitmap	BITMAP64	0x1f
Unsigned 8-bit int	UINT8	0x20
Unsigned 16-bit int	UINT16	0x21
Unsigned 24-bit int	UINT24	0x22
Unsigned 32-bit int	UINT32	0x23
Unsigned 40-bit int	UINT40	0x24
Unsigned 48-bit int	UINT48	0x25
Unsigned 56-bit int	UINT56	0x26
Unsigned 64-bit int	UINT64	0x27
Signed 8-bit int	INT8	0x28
Signed 16-bit int	INT16	0x29
Signed 24-bit int	INT24	0x2a
Signed 32-bit int	INT32	0x2b
Signed 40-bit int	INT40	0x2c
Signed 48-bit int	INT48	0x2d
Signed 56-bit int	INT56	0x2e
Signed 64-bit int	INT64	0x2f
8-bit enumeration	ENUM8	0x30
16-bit enumeration	ENUM16	0x31
Semi-precision	FLOAT2	0x38
Single precision	FLOAT4	0x39
Double precision	FLOAT8	0x3a
Octet String	STRING_OCTET	0x41
Character String	STRING_CHAR	0x42
Long Octet String	STRING_LONG_OCTET	0x43
Long Character String	STRING_LONG_CHAR	0x44
Array	ARRAY	0x48
Structure	STRUCTURE	0x4c
Set	SET	0x50
Bag	BAG	0x51
Time of day	TIME_OF_DAY	0xe0
Date	DATE	0xe1
UTCTime	UTCTIME	0xe2
Cluster ID	CLUSTER_ID	0xe8
Attribute ID	ATTRIBUTE_ID	0xe9
BACnet OID	BACNET_OID	0xea
IEEE address	IEEE_ADDRESS	0xf0
128-bit security key	SECKEY128	0xf1
Unknown	UNKNOWN	0xff

See the [ZigBee Cluster Library \(ZCL\)](#) specification for more information on the different data types and how they are used.

DataType.getLength()

This method is used to get the length of a variable of the given type. This length is in number of bytes. For variable length or unknown types it will return `null`

Signature:

```
DataType.isVariableLength(type)
```

Parameters:

- **type:** The type to check. This should be one of the *DataType Constants* (page 1056) defined above.

Example

```
DataType.getLength(DataType.UINT8) // returns 1
DataType.getLength(DataType.CLUSTER_ID) // returns 2
DataType.getLength(DataType.STRING_CHAR) // returns null
```

DataType.isVariableLength()

This method is used to test if a given type is variable length or not.

Signature:

```
DataType.isVariableLength(type)
```

Parameters:

- **type:** The type to check. This should be one of the *DataType Constants* (page 1056) defined above.

Example

```
DataType.isVariableLength(DataType.UINT8) // returns false
DataType.isVariableLength(DataType.STRING_CHAR) // returns true
```

DataType.isDiscrete()

This method is used to test if a given type is discrete.

Signature:

```
DataType.isDiscrete(type)
```

Parameters:

- **type:** The type to check. This should be one of the *DataType Constants* (page 1056) defined above.

Example

```
DataType.isDiscrete(DataType.UINT8) // returns true
DataType.isDiscrete(DataType.FLOAT2) // returns false
```

DataType.pack()

This method is used to pack data of a given type into hex string form. Currently not all DataTypes are supported by this method. All discrete data types are supported, but only `STRING_CHAR` is supported for variable length data types. If the type passed in is `null`, an empty string, or `DataType.UNKNOWN` the value of data will be returned unmodified.

Signature:

```
DataType.pack(data, type, littleEndian=false)
```

Parameters:

- **data:** The data to pack, the type of this should be appropriate for the type argument
- **type:** The type of the data being packed. This should be one of the *DataType Constants* (page 1056) defined above.
- **littleEndian:** If true it will pack it with the least significant bits first.

Example

```
DataType.pack(0x01, DataType.UINT8) // returns "01"  
DataType.pack(0x01, DataType.UINT64) // returns "0000000000000001"  
DataType.pack(0x01, DataType.UINT32, true) // returns "01000000"
```

284.19 Z-Wave Reference

You can find the reference documentation for the SmartThings Z-Wave library [here](#) (requires login).

The Z-Wave public specification can be found [here](#).

Part XXI

Contributing to the Docs

Writing Style Guide

When you write for SmartThings platform, your audience should find your documentation readable, interesting and informative. To accomplish these goals, we encourage you to stick to our recommended writing style.

285.1 Titles and headings

Wherever possible, write purpose-driven documentation. This means writing document titles and section headings that state the benefit explicitly. Such titles or headings can be written as either calls to actions or tasks that can be done. This approach makes it easier for the reader to learn how to get her job done.

Examples:

- Preferred: Writing Your First SmartApp (document title)
- Avoid: SmartApp Fundamentals (document title isn't purpose-driven)
- Preferred: Create your own RESTful API (section heading)
- Avoid: Parent-child SmartApps (document title isn't purpose-driven)
- Preferred: Combine Multiple Automations (document title)

Note:

- A **document title** is the main title of a document page. A document has only one document title. Example: “Writing Style Guide” at the beginning of this page. The document title also appears at the top level in the navigation bar, so it must be short, preferably four to five words or less.
 - A **section heading** is the title for an individual section within a document page. Example: “Titles and headings” at the top of this section. A document page can have multiple sections, and hence multiple section headings.
-

285.1.1 Avoid framing as questions

Avoid using questions in document titles and section headings.

Example:

- Avoid: How does the switch turn on?
- Preferred: How the switch turns on (section heading)

285.1.2 Avoid italics (emphasis)

Avoid using emphasis (italics) in document titles or section headings. See *Page structure* (page 1066).

285.1.3 Document titles

Use title case, as in “Document Titles” and not “Document titles.”

Example:

- The title of this document, “Writing Style Guide.”

285.1.4 What not to capitalize in title

- as (SmartApp as a Wakeup Device)
- to (How to Subscribe to Events)
- on (Trigger on Time)
- at (Alarm at a Specific Time)
- of, the (No More Rules of the Game)
- in (Your First SmartApp in Five Minutes)
- the (Motion at the Garage Door)
- a (Three Critical Triggers in a Day)
- for (Temperature Control for Basement)

285.1.5 What to capitalize in title

- is (Device Health Is Reported Weekly)
- was (Motion Was Detected)
- then (Set a Sunrise Automation, Then Sit Back)
- up (Biff Stood Up to Trigger the Alarm)
- that (The Battery That Lasts Forever)
- with (The Day Ends With Cheers All Around)

285.1.6 Section headings

Use sentence case, as in “Subscription management,” and not the title case, as in “Subscription Management.”

Note:

- A **sentence case** is where you only capitalize the first letter of the sentence.
 - A **title case** is where you capitalize first letter of every word of the sentence.
-
-

285.2 UI elements

Always use italics emphasis when quoting a UI element label such as a button label or an icon label.

Example:

Go to the *Simulator* menu, and click on *Browse SmartApp Templates* in the dropdown list.

Here it is in reStructuredText:

Go to the **Simulator** menu, and click on **Browse SmartApp Templates** in the dropdown list.

285.3 List elements

Always start a list segment with a heading and a colon.

Example:

To publish a SmartApp or a Device Handler for yourself, follow these steps:

- Make sure that you are in the proper Location.
 - From your SmartApp or Device Handler view, click on *Publish* button. Then click the *For Me* option.
-

If it is a complete sentence, always end the list element, numbered or unordered, with a period.

Note: This applies also for a list element that has multiple sentences.

Example:

To publish a SmartApp or a Device Handler for yourself, follow these steps:

- Make sure that you are in the proper Location.
 - From your SmartApp or Device Handler view, click on *Publish* button. Then click the *For Me* option.
-

If it is an incomplete sentence, do not end the list element with a period.

Example:

When you finish this tutorial, you will know:

- Key components of a SmartApp
 - Features of IDE
 - Controlling devices
-

Always write a list sentence in the sentence case.

Example:

- (YES) Make sure that you are in the proper Location.
 - (NO) Make Sure That You Are In the Proper Location.
-

Avoid more than two levels of lists.

Example:

(YES) SmartThings platform supports various Hub scenarios such as:

- **There may not be a hub at all**
 - There may be a third-party Hub present
 - An all-cloud environment with no Hub whatsoever
- SmartApps may run across both cloud and Hub connected devices
- There may be multiple Hubs

(NO) SmartThings platform supports various Hub scenarios such as:

- **There may not be a hub at all**
 - **There may be a third-party Hub present**
 - * Highlight supported third-party Hubs
 - An all-cloud environment with no Hub whatsoever
 - SmartApps may run across both cloud and Hub connected devices
 - There may be multiple Hubs
-

285.4 Page structure

Each document should be named with a `.rst` file extension. Each page is composed of a title, followed by some short text outlining the purpose of the document.

Sections should be delimited by `----`, to insert a line separator.

The structure should look like this:

```
=====  
Page Title  
=====  
  
Some introductory material.  
  
----  
  
Section 1  
-----  
  
Section text.  
  
----  
  
Section 2  
-----
```

```
Section text.  
  
Subsection 2.1  
^^^^^^^^^^^^^^  
  
Subsection text.
```

285.4.1 Page title

Page titles appear at the top of the document, and have a row of === characters above and below. Page titles should have title capitalization:

```
=====  
This is a Page Title  
=====
```

285.4.2 Headings

Top-level section headings are followed by a row of --- characters. They should have sentence capitalization:

```
This is a section  
-----
```

Subsection headings are followed by a row of ^^ characters. They should have sentence capitalization.

```
This is a section  
-----  
  
This is a subsection  
^^^^^^^^^^^^^^
```

Note: Not all documents currently follow the guideline of using ^^ for subsections. If you are editing a document and see a different heading syntax, feel free to change it.

285.5 reStructuredText syntax

285.5.1 Links

Links to external targets look like this:

```
`SmartThings <http://smarththings.com>`_
```

Links to sections within the document can be included like this:

```
Section name  
-----  
  
See `Other section`_ for more information.  
  
Other section  
-----
```

The `:ref:` target allows us to link to other documents or document sections. It requires placing a label above a section, title, or image:

```
.. _section_label:  
  
Some section  
-----
```

Another document can then link to `Some section` like this:

```
See :ref: section_label for more information.
```

285.5.2 Lists

Ordered lists appear like this:

```
#. Item 1  
#. Item 2  
#. Item 3
```

Which results in:

1. Item 1
2. Item 2
3. Item 3

Unordered lists use a `-` or `*` character:

```
- First bullet  
- Second bullet
```

285.5.3 Inline markup

- Surround text with `*` for *italics text*.
- Surround text with `**` for **strong text**.
- Surround text with `“` for code samples (`someMethod()`).

When referring to method calls in the documentation, place `()` after the method name: `methodName()`. This helps distinguish methods from other code literals.

285.5.4 Code examples

Code blocks can be included using the `code-block` directive. Use the appropriate language for the code sample. Code blocks may appear with line numbers (use `:linenos:`) and may emphasize certain lines:

```
.. code-block:: groovy  
   :linenos:  
   :emphasize-lines: 3  
  
   def someMethod() {  
       def myVar = 14  
       doSomethingAmazing(myVar)  
   }
```


The above code block renders as:

```

1 def someMethod() {
2     def myVar = 14
3     doSomethingAmazing(myVar)
4 }

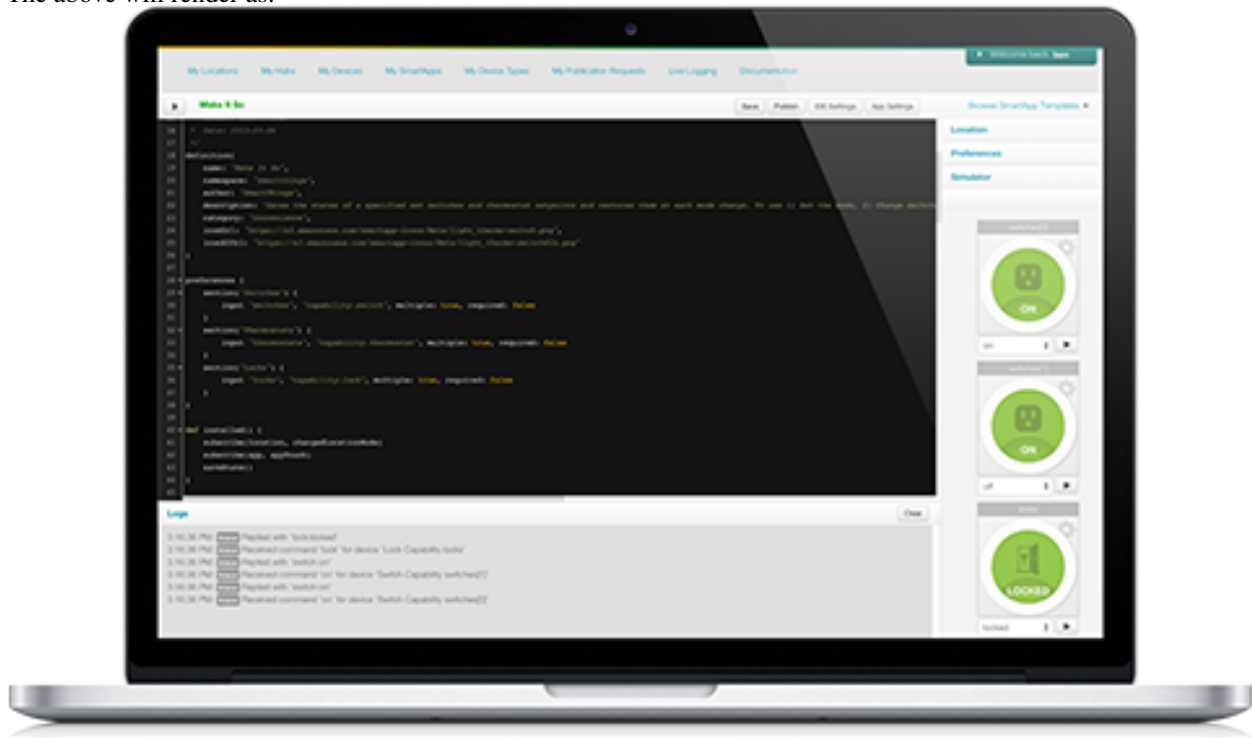
```

285.5.5 Images

Images are found in the `/img` directory of the documentation, and can be included like this (you may need to alter the path depending on the location of the document):

```
... image:: ../img/getting-started/building-img.png
```

The above will render as:



285.5.6 Admonitions

Admonitions are ways of calling out certain bodies of text:

```
... note::
```

A note provides more information about the content, in a side-bar like format.

```
... tip::
```

A tip is some extra information that while not strictly necessary, may lead to the reader learning

```
... warning::
```

```
A warning is just that - a warning of something that the reader should be aware of.  
... error:  
An error is for error conditions.
```

The above results in:

Note: A note provides more information about the content, in a side-bar like format.

Tip: A tip is some extra information that while not strictly necessary, may lead to the reader learning a new way of doing something.

Warning: A warning is just that - a warning of something that the reader should be aware of.

Error: An error is for error conditions.

285.5.7 Tables

Simple tables in RST look like this:

```
=====  
Heading 1 Heading 2  
=====  
1.1      1.2  
2.1      2.2  
=====
```

The above renders as:

Heading 1	Heading 2
1.1	1.2
2.1	2.2

Grid tables can be written like this:

```
+-----+-----+-----+  
| Header 1 | Header 2 | Header 3 |  
+-----+-----+-----+  
| body row 1 | column 2 | column 3 |  
+-----+-----+-----+  
| body row 2 | Cells may span columns.|  
+-----+-----+-----+  
| body row 3 | Cells may | - Cells |  
+-----+ span rows. | - contain |  
| body row 4 | | - blocks. |  
+-----+-----+-----+
```

Which results in:

Header 1	Header 2	Header 3
body row 1	column 2	column 3
body row 2	Cells may span columns.	
body row 3	Cells may span rows.	
body row 4		

Cells

contain

285.6 API reference documents

blocks.

The API reference documentation contains all public API method definitions. API reference documentation is located in the `ref-docs/` directory.

285.6.1 Organization

API reference documents include an introduction and a listing of all APIs in alphabetical order.

Note: The SmartApp and Device Handler API reference documentation lists all required callback methods to be listed first. The remaining APIs are then listed in alphabetical order.

285.6.2 Introduction

Each API reference document contains a brief overview of the API, along with a quick example of how to reference the object (if applicable).

Consider the example of the Device API reference documentation:

```

=====
Device
=====

The Device object represents a physical device in a SmartApp.
When a user installs a SmartApp, they typically will select the devices to be used by the SmartApp.
SmartApps can then interact with these Device objects to get device information, or send commands to

Device objects cannot be instantiated, but are created by the SmartThings platform and available via

... code-block: groovy

    preferences {
        section() {
            // prompt user to select a device that supports the switch capability.
            // assign the chosen device to a variable named "theswitch"
            input "theswitch", "capability.switch"
        }
    }
    ...
    // access Device instance using the input name:
    def deviceDisplayName = theswitch.displayName
    ...

```

285.6.3 Method documentation

Method documentation adheres to these rules:

- The method name is a first-level heading followed by an open and close parentheses (to denote it is a method, not a property).
- A brief description of the method follows the first-level heading.
- The method's signature, parameters, return type, any declared exceptions, and a brief example follows.

The example below illustrates this, and can be used as a template when writing API documentation. Each component title (Signature, Parameters, etc.) of the API documentation is bolded, and the content follows on the next line, indented by one tab (or four spaces). Details about each component follows.

```
rgbToHex()
-----

Converts an RGB value to a hexadecimal color string.

**Signature:**
    ``static String rgbToHex(red, green, blue) throws IllegalArgumentException``

**Parameters:**
    `Integer`_ red - The red value, between 0 and 255

    `Integer`_ green - The green value, between 0 and 255

    `Integer`_ blue - The blue value, between 0 and 255

**Returns:**
    `String`_ - The hexadecimal representation of the RGB value

**Throws:**
    `IllegalArgumentException`_ - An ``IllegalArgumentException`` is thrown if any of the RGB values

**Example:**

.. code-block:: groovy

    def deepSkyBlueInHex = colorUtil.rgbToHex(0, 191, 255)
    log.debug "RGB 0,191,255 in Hex is $deepSkyBlueInHex"
```

Signature

The method signature is the same as the method's source definition, formatted as an inline code block.

Parameters

Method parameters are documented according to the following rules:

- Each parameter is listed, in order, with a link to the return type.
- All external links are defined at the bottom of the document.
- In cases of standard Java return types, a link to the Java 7 JavaDocs for the type is used. If the return type is a SmartThings object, a link to that SmartThings object reference document is used.
- If the method does not accept parameters, the entire parameters block is omitted.

- Optional parameters are placed inside square brackets.
- Parameters that accept a map include a table listing all the supported key/value pairs:

```

**Signature:**
  ``List<Event> events([max: N])``

**Parameters:**
  ``Map``_ options *(optional)* - Options for the query. Supported options:

  =====
  option  Type          Description
  =====
  ``max``  ``Number``_ The maximum number of Events to return. By default, the maximum is 10.
  =====

```

Returns

Method return values are documented according to the following rules:

- The return statement begins with a link to the return type (external or internal), along with a brief description of the value returned.
- In the case of `void` return types, do not include the “Returns” component.

For example:

```

**Returns:**
  ``String``_ - The hexadecimal representation of the RGB value

```

Throws

Methods that throw an exception as part of their contract include a “Throws” component, with a link to the exception type, and when the exception is thrown:

```

**Throws:**
  ``IllegalArgumentException``_ - An ``IllegalArgumentException`` is thrown if any of the RGB values

```

Example

Most methods include an example of their usage. The example code should include the minimum amount of code to highlight the documented method.

Some simple methods may not require an example—use your judgement.

285.7 Miscellaneous tips

- Spell check before committing.
- Show, don’t tell - include example code.
- Place each sentence on a new line to help with review and readability.

- Not all documents currently follow these guidelines. See the [Contributing](#) guide to learn how you can contribute, and help address that. :)
-

285.8 SmartThings glossary

Recommended style	Not recommended
Cloud-connected	cloud-connected, cloud connected, Cloud connected
Composite Device	CompositeDevice, Composite device, composite device
Contact Book	contact book
Contacts	contacts
Device Handler	Device handler, DeviceHandler, Device Type Handler, device handler, devicehandler
editor	Editor
Event	event
event handler	Event Handler, Event handler
Hub	hub
“Hub version 2” first time, then “Hub v2”	Hub v 2, Hub v.2, Hub V2.
“Internet of Things” first time, then “IoT”	IOT
internet	Internet
LAN-connected	lan-connected, lan connected, LAN connected
Location	location
Marketplace	Market place, Market Place, MarketPlace
Mode	mode
My Home	MyHome, myHome, My home
Routines	routines, Hello Home actions
Samsung SmartThings Hub	SamsungSmartThings Hub, Samsung SmartThings hub
Simulator	simulator
smart home	SmartHome, Smart Home, smarthome
Smart Home Monitor	SMH, smarthome monitor, SmartHome monitor
SmartApp	Smart app, Smart App, Smartapp, smartapp, smart app
SmartThings	Smart Things, Smartthings, Smart things
Welcome Code	Welcome code, WelcomeCode, Claim code, ClaimCode
Z-Wave	ZWave, Z-wave
ZigBee	Zigbee, Zig Bee

285.9 Further reading

- [Sphinx documentation](#)
- [reStructuredText Reference](#)

Part XXII

Samsung SmartThings Hub FAQ

A collection of frequently asked questions about the new Samsung SmartThings Hub, and updated SmartThings mobile applications, for SmartThings developers.

What can developers do with the new Samsung SmartThings Hub and updated mobile apps?

Developers can use the new Contact Book feature to easily send notifications to a user's selected contacts, without requesting the user to enter in a phone number for each SmartApp. You can learn more about it in the [Sending Notifications](#) (page 385) documentation.

The new iOS and Android mobile apps also make use of a new Device Tiles layout, that uses a 6 column grid. There is also a new tile available to use for devices - multi-attribute tiles allow a single tile to display information about more than one attribute of a device. You can learn more in the [Tiles](#) (page 473) documentation.

With the new Hub and mobile experience, we've also laid the groundwork for exciting new developer features in the near future. Our developers are what makes SmartThings great, and we're excited to build together!

Do I need to update my SmartApps or Device Handlers?

Most custom SmartApps and Device Types will continue to work without the need for code changes. There are some features that you may wish to take advantage of, however, like the new multi-attribute device tile. SmartApps that send notifications should be updated to use the new Contact Book feature, but they will continue to function as they did before without updating your code.

Despite our best intentions and precautions, it is possible that your custom SmartApp or Device Type may not work as it did before. If this is the case, please report the issue to support at support@smarthings.com (include example code, relevant log messages, and screenshots if applicable). The [SmartThings Community Forums](#) are also a good place for developers to help one another. The SmartThings Community Team will be monitoring the forums to identify and help with issues, and incorporating feedback into our documentation.

Hello Home Actions now appear as "Routines" in the mobile application. Do I need to update any of my SmartApps to get or execute Routines?

No. SmartApps that work with Routines still use the methods discussed in the [Routines](#) (page 339) documentation.

At some point in the future, we may create new methods that reflect the terminology change, but we will not do so without advanced notification.

How does local SmartApp or Device Type processing work?

Certain automations can now execute locally on the Samsung SmartThings Hub. The SmartThings internal team specifies which automations are eligible for local execution. This process requires evaluation and testing of the SmartApp and devices, as well as ensuring that the necessary code artifacts are delivered to the Hub.

Any locally executing SmartApps or Device Handlers still send events to the SmartThings cloud. This is necessary so that the mobile application can accurately reflect the current state of the devices, as well as perform any cloud-required services (e.g., sending notifications). In the event of an internet outage, the events will be queued and sent to the SmartThings cloud when internet is restored.

It is not possible for developers to specify that certain Device Types or SmartApps execute in any particular location (cloud or on the Hub). SmartApps or Device Types that have not been reviewed, tested, and delivered to the Hub by the SmartThings team will execute in the SmartThings cloud.

What happens when the internet to the Hub goes out?

Provided there is still power to the Hub (wired or battery), any SmartApps that are able to execute locally will still run without an internet connection. The mobile app will report the Hub is offline, and because there are no events being sent to the SmartThings cloud, notifications will not work.

The radios in the Hub will still function without internet. Events to the cloud will be queued, and sent when the internet is restored.

The mobile app has some new video-related features. How can developers utilize those capabilities?

The APIs for working with the new video features are not yet available, but we are excited to bring them to you soon!

Does the Hub have UDP support?

UDP access for developers is not currently supported, but may come in future updates.

Does the Hub support local file storage?

The Samsung SmartThings Hub stores some information about SmartApps, Device Types, and Locations locally, but this is not publicly accessible.

Can I SSH into the Hub?

No, you cannot SSH into the Samsung SmartThings Hub.

What about Bluetooth?

The Samsung SmartThings Hub ships with BLE to support future expandability, but will be inactive at product launch.

What can I do with the USB ports?

Adding USB ports to the Samsung SmartThings Hub allows for future expandability, but will have no functionality at product launch.

Does the Hub support IPv6?

No. This may come in future updates.

Does the Hub support WebSocket or Telnet for developers?

The Samsung SmartThings Hub does not support WebSocket, Telnet, or raw socket access for developers.

Does the Hub support getting local device status, or controlling local devices, without going through the SmartThings cloud? For example, can I just access the Hub to get device status or control devices?

Currently, no. We know this is a requested feature, and have identified it for future roadmap consideration.
