salt-api Documentation

Release 0.8.4

Thomas Hatch

December 07, 2016

Contents

1 Getting started 3

2 Installation quickstart 5

3 `netapi` modules 7

4 Releases 33

5 Reference 41

HTTP Routing Table 43

Python Module Index 45

Warning: Outdated documentation
The `salt-api` project has been merged into the main Salt repository as of Salt's **Helium** release.
No api changes were made. The `salt-api` daemon remains intact and is now available in the default `salt-master` install.
The documentation has been moved into the main salt project as well. `netapi` module documentation is available in the module index.
No further development will take place in this repository. It will be left in the current state for historical purposes.
Open issues will be migrated to the Salt repository.
Current documentation now lives within the main Salt documentation.

- Introduction to netapi modules
- Full list of netapi modules
- Archived release notes

The documentation for the final salt-api release, 0.8.4.1, is included below.

`salt-api` is a modular interface on top of Salt that can provide a variety of entry points into a running Salt system. It can start and manage multiple interfaces allowing a REST API to coexist with XMLRPC or even a Websocket API.

CHAPTER 1

Getting started

1. Install **salt-api** on the same machine as your Salt master.
2. Edit your Salt master config file for all required options for each `netapi` module you wish to run.
3. Install any required additional libraries or software for each `netapi` module you wish to run.
4. Run **salt-api** which will then start all configured `netapi` modules.

Note: Each `netapi` module will have differing configuration requirements and differing required software libraries.

Exactly like the various module types in Salt (execution modules, renderer modules, returner modules, etc.), netapi modules in **salt-api** will *not* be loaded into memory or started if all requirements are not met.

CHAPTER 2

Installation quickstart

2.1 salt-api Quickstart

salt-api manages netapi modules which are modules that (usually) bind to a port and start a service. Each netapi module will have specific requirements for third-party libraries and configuration (which goes in the master config file). Read the documentation for each netapi module to determine what is needed.

For example, the *rest_cherrypy* netapi module requires that CherryPy be installed and that a `rest_cherrypy` section be added to the master config that specifies which port to listen on.

2.1.1 Installation

PyPI

https://pypi.python.org/pypi/salt-api

```
pip install salt-api
```

RHEL, Fedora, CentOS

RPMs are available in the Fedora repositories and EPEL:

```
yum install salt-api
```

Ubuntu

PPA packages available for Ubuntu on LaunchPad:

```
sudo add-apt-repository ppa:saltstack/salt
sudo apt-get update
sudo apt-get install salt-api
```

openSUSE, SLES

RPMs are available via the OBS:

```
zypper install salt-api
```

CHAPTER 3

netapi modules

The core functionality for **salt-api** lies in pluggable netapi modules that adhere to the simple interface of binding to a port and starting a service. **salt-api** can manage one or many services concurrently.

3.1 Full list of netapi modules

3.1.1 Full list of netapi modules

rest_cherrypy

A REST API for Salt

depends

- CherryPy Python module

optdepends

- ws4py Python module for websockets support.

configuration All authentication is done through Salt's external auth system which requires additional configuration not described here.

Example production-ready configuration; add to the Salt master config file:

```
rest_cherrypy:
  port: 8000
  ssl_crt: /etc/pki/tls/certs/localhost.crt
  ssl_key: /etc/pki/tls/certs/localhost.key
```

Using only a secure HTTPS connection is strongly recommended since Salt authentication credentials will be sent over the wire.

A self-signed certificate can be generated using the create_self_signed_cert() function in Salt (note the dependencies for this module).

```
% salt-call tls.create_self_signed_cert
```

All available configuration options are detailed below. These settings configure the CherryPy HTTP server and do not apply when using an external server such as Apache or Nginx.

port **Required**

The port for the webserver to listen on.

host [`0.0.0.0`] The socket interface for the HTTP server to listen on.

New in version 0.8.2.

debug [`False`] Starts the web server in development mode. It will reload itself when the underlying code is changed and will output more debugging info.

ssl_crt The path to a SSL certificate. (See below)

ssl_key The path to the private key for your SSL certificate. (See below)

disable_ssl A flag to disable SSL. Warning: your Salt authentication credentials will be sent in the clear!

New in version 0.8.3.

webhook_disable_auth [False] The *`Webhook`* URL requires authentication by default but external services cannot always be configured to send authentication. See the Webhook documentation for suggestions on securing this interface.

New in version 0.8.4.1.

webhook_url [/hook] Configure the URL endpoint for the *`Webhook`* entry point.

New in version 0.8.4.1.

thread_pool [`100`] The number of worker threads to start up in the pool.

Changed in version 0.8.4: Previous versions defaulted to a pool of `10`

socket_queue_size [`30`] Specify the maximum number of HTTP connections to queue.

Changed in version 0.8.4: Previous versions defaulted to `5` connections.

max_request_body_size [`1048576`] Changed in version 0.8.4: Previous versions defaulted to `104857600` for the size of the request body

collect_stats [False] Collect and report statistics about the CherryPy server

New in version 0.8.4.

Reports are available via the *`Stats`* URL.

static A filesystem path to static HTML/JavaScript/CSS/image assets.

static_path [`/static`] The URL prefix to use when serving static assets out of the directory specified in the `static` setting.

New in version 0.8.2.

app A filesystem path to an HTML file that will be served as a static file. This is useful for bootstrapping a single-page JavaScript app.

New in version 0.8.2.

app_path [`/app`] The URL prefix to use for serving the HTML file specified in the `app` setting. This should be a simple name containing no slashes.

Any path information after the specified path is ignored; this is useful for apps that utilize the HTML5 history API.

New in version 0.8.2.

root_prefix [/] A URL path to the main entry point for the application. This is useful for serving multiple applications from the same URL.

New in version 0.8.4.

Authentication Authentication is performed by passing a session token with each request. Tokens are generated via the *Login* URL.

The token may be sent in one of two ways:

- Include a custom header named *X-Auth-Token*.
- Sent via a cookie. This option is a convenience for HTTP clients that automatically handle cookie support (such as browsers).

See also:

You can bypass the session handling via the *Run* URL.

Usage Commands are sent to a running Salt master via this module by sending HTTP requests to the URLs detailed below.

Content negotiation

This REST interface is flexible in what data formats it will accept as well as what formats it will return (e.g., JSON, YAML, x-www-form-urlencoded).

- Specify the format of data in the request body by including the *Content-Type* header.
- Specify the desired data format for the response body with the *Accept* header.

Data sent in POST and PUT requests must be in the format of a list of lowstate dictionaries. This allows multiple commands to be executed in a single HTTP request.

lowstate A dictionary containing various keys that instruct Salt which command to run, where that command lives, any parameters for that command, any authentication credentials, what returner to use, etc.

Salt uses the lowstate data format internally in many places to pass command data between functions. Salt also uses lowstate for the LocalClient() Python API interface.

The following example (in JSON format) causes Salt to execute two commands:

```
[{
    "client": "local",
    "tgt": "*",
    "fun": "test.fib",
    "arg": ["10"]
},
{
    "client": "runner",
    "fun": "jobs.lookup_jid",
    "jid": "20130603122505459265"
}]
```

---

x-www-form-urlencoded

Sending JSON or YAML in the request body is simple and most flexible, however sending data in urlencoded format is also supported with the caveats below. It is the default format for HTML forms, many JavaScript libraries, and the **curl** command.

For example, the equivalent to running `salt '*' test.ping` is sending `fun=test.ping&arg&client=local&tgt=*` in the HTTP request body.

Caveats:

- Only a single command may be sent per HTTP request.
- Repeating the `arg` parameter multiple times will cause those parameters to be combined into a single list.

  Note, some popular frameworks and languages (notably jQuery, PHP, and Ruby on Rails) will automatically append empty brackets onto repeated parameters. E.g., `arg=one`, `arg=two` will be sent as `arg[]=one`, `arg[]=two`. This is not supported; send JSON or YAML instead.

Deployment

The `rest_cherrypy` netapi module is a standard Python WSGI app. It can be deployed one of two ways.

`salt-api` using the CherryPy server The default configuration is to run this module using **`salt-api`** to start the Python-based CherryPy server. This server is lightweight, multi-threaded, encrypted with SSL, and should be considered production-ready.

Using a WSGI-compliant web server This module may be deplayed on any WSGI-compliant server such as Apache with mod_wsgi or Nginx with FastCGI, to name just two (there are many).

Note, external WSGI servers handle URLs, paths, and SSL certs directly. The `rest_cherrypy` configuration options are ignored and the `salt-api` daemon does not need to be running at all. Remember Salt authentication credentials are sent in the clear unless SSL is being enforced!

An example Apache virtual host configuration:

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias *.example.com

    ServerAdmin webmaster@example.com

    LogLevel warn
    ErrorLog /var/www/example.com/logs/error.log
    CustomLog /var/www/example.com/logs/access.log combined

    DocumentRoot /var/www/example.com/htdocs

    WSGIScriptAlias / /path/to/saltapi/netapi/rest_cherrypy/wsgi.py
</VirtualHost>
```

REST URI Reference

- */*
- */login*
- */logout*
- */minions*
- */jobs*
- */run*
- */events*
- */ws*
- */hook*
- */stats*

/

class saltapi.netapi.rest_cherrypy.app.**LowDataAdapter**

The primary entry point to Salt's REST API

GET()

An explanation of the API with links of where to go next

GET /

Request Headers

- Accept – the desired response format.

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

Example request:

```
% curl -i localhost:8000
```

```
GET / HTTP/1.1
Host: localhost:8000
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

POST

Send one or more Salt commands in the request body

POST /

Request Headers

- *X-Auth-Token* – a session token from *Login*.
- Accept – the desired response format.
- Content-Type – the format of the request body.

Response Headers

- Content-Type – the format of the response body; depends on the *Accept* request header.

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

lowstate data describing Salt commands must be sent in the request body.

Example request:

```
% curl -si https://localhost:8000 \
        -H "Accept: application/x-yaml" \
        -H "X-Auth-Token: d40d1e1e" \
        -d client=local \
        -d tgt='*' \
        -d fun='test.ping' \
        -d arg
```

```
POST / HTTP/1.1
Host: localhost:8000
Accept: application/x-yaml
X-Auth-Token: d40d1e1e
Content-Length: 36
Content-Type: application/x-www-form-urlencoded

fun=test.ping&arg&client=local&tgt=*
```

Example response:

```
HTTP/1.1 200 OK
Content-Length: 200
Allow: GET, HEAD, POST
Content-Type: application/x-yaml

return:
- ms-0: true
    ms-1: true
    ms-2: true
    ms-3: true
    ms-4: true
```

/login

class saltapi.netapi.rest_cherrypy.app.**Login**(**args*, ***kwargs*)

Log in to recieve a session token

Authentication information.

GET()

Present the login interface

GET /login

An explanation of how to log in.

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

Example request:

```
% curl -i localhost:8000/login
```

```
GET /login HTTP/1.1
Host: localhost:8000
Accept: text/html
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: text/html
```

POST (***kwargs*)

Authenticate against Salt's eauth system

Changed in version 0.8.0: No longer returns a 302 redirect on success.

Changed in version 0.8.1: Returns 401 on authentication failure

POST /login

Request Headers

- *X-Auth-Token* – a session token from *Login*.
- Accept – the desired response format.
- Content-Type – the format of the request body.

Form Parameters

- **eauth** – the eauth backend configured for the user
- **username** – username
- **password** – password

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

Example request:

```
% curl -si localhost:8000/login \
        -H "Accept: application/json" \
        -d username='saltuser' \
        -d password='saltpass' \
        -d eauth='pam'
```

```
POST / HTTP/1.1
Host: localhost:8000
Content-Length: 42
Content-Type: application/x-www-form-urlencoded
Accept: application/json

username=saltuser&password=saltpass&eauth=pam
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 206
X-Auth-Token: 6d1b722e
Set-Cookie: session_id=6d1b722e; expires=Sat, 17 Nov 2012 03:23:52 GMT; Path=/

{"return": {
    "token": "6d1b722e",
    "start": 1363805943.776223,
    "expire": 1363849143.776224,
    "user": "saltuser",
    "eauth": "pam",
    "perms": [
        "grains.*",
        "status.*",
        "sys.*",
        "test.*"
```

```
            ]
        }}
```

/logout
class saltapi.netapi.rest_cherrypy.app.**Logout**

POST()
Destroy the currently active session and expire the session cookie

New in version 0.8.0.

/minions
class saltapi.netapi.rest_cherrypy.app.**Minions**

GET(*mid=None*)
A convenience URL for getting lists of minions or getting minion details

GET /minions/(*mid*)

Request Headers

- *X-Auth-Token* – a session token from *Login*.
- Accept – the desired response format.

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

Example request:

```
% curl -i localhost:8000/minions/ms-3
```

```
GET /minions/ms-3 HTTP/1.1
Host: localhost:8000
Accept: application/x-yaml
```

Example response:

```
HTTP/1.1 200 OK
Content-Length: 129005
Content-Type: application/x-yaml

return:
- ms-3:
    grains.items:
        ...
```

POST(***kwargs*)
Start an execution command and immediately return the job id

POST /minions

Request Headers

- *X-Auth-Token* – a session token from *Login*.
- Accept – the desired response format.
- Content-Type – the format of the request body.

Response Headers

- Content-Type – the format of the response body; depends on the *Accept* request header.

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

lowstate data describing Salt commands must be sent in the request body. The `client` option will be set to `local_async()`.

Example request:

```
% curl -sSi localhost:8000/minions \
    -H "Accept: application/x-yaml" \
    -d tgt='*' \
    -d fun='status.diskusage'
```

```
POST /minions HTTP/1.1
Host: localhost:8000
Accept: application/x-yaml
Content-Length: 26
Content-Type: application/x-www-form-urlencoded

tgt=*&fun=status.diskusage
```

Example response:

```
HTTP/1.1 202 Accepted
Content-Length: 86
Content-Type: application/x-yaml

return:
- jid: '20130603122505459265'
    minions: [ms-4, ms-3, ms-2, ms-1, ms-0]
_links:
    jobs:
    - href: /jobs/20130603122505459265
```

/jobs

class saltapi.netapi.rest_cherrypy.app.**Jobs**

GET (*jid=None*)

A convenience URL for getting lists of previously run jobs or getting the return from a single job

GET /jobs/ (*jid*)

List jobs or show a single job from the job cache.

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

Example request:

```
% curl -i localhost:8000/jobs
```

```
GET /jobs HTTP/1.1
Host: localhost:8000
Accept: application/x-yaml
```

Example response:

```
HTTP/1.1 200 OK
Content-Length: 165
Content-Type: application/x-yaml

return:
- '20121130104633606931':
    Arguments:
    - '3'
    Function: test.fib
    Start Time: 2012, Nov 30 10:46:33.606931
    Target: jerry
    Target-type: glob
```

Example request:

```
% curl -i localhost:8000/jobs/20121130104633606931
```

```
GET /jobs/20121130104633606931 HTTP/1.1
Host: localhost:8000
Accept: application/x-yaml
```

Example response:

```
HTTP/1.1 200 OK
Content-Length: 73
Content-Type: application/x-yaml

info:
- Arguments:
    - '3'
    Function: test.fib
    Minions:
    - jerry
    Start Time: 2012, Nov 30 10:46:33.606931
    Target: '*'
    Target-type: glob
    User: saltdev
    jid: '20121130104633606931'
return:
- jerry:
    - - 0
    - 1
    - 1
    - 2
    - 6.9141387939453125e-06
```

/run
class saltapi.netapi.rest_cherrypy.app.**Run**

POST (***kwargs*)
Run commands bypassing the *normal session handling*

New in version 0.8.0.

POST /run
This entry point is primarily for "one-off" commands. Each request must pass full Salt authentication

credentials. Otherwise this URL is identical to the *root URL (/)*.

lowstate data describing Salt commands must be sent in the request body.

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

Example request:

```
% curl -sS localhost:8000/run \
    -H 'Accept: application/x-yaml' \
    -d client='local' \
    -d tgt='*' \
    -d fun='test.ping' \
    -d username='saltdev' \
    -d password='saltdev' \
    -d eauth='pam'
```

```
POST /run HTTP/1.1
Host: localhost:8000
Accept: application/x-yaml
Content-Length: 75
Content-Type: application/x-www-form-urlencoded

client=local&tgt=*&fun=test.ping&username=saltdev&password=saltdev&eauth=pam
```

Example response:

```
HTTP/1.1 200 OK
Content-Length: 73
Content-Type: application/x-yaml

return:
- ms-0: true
    ms-1: true
    ms-2: true
    ms-3: true
    ms-4: true
```

/events

class saltapi.netapi.rest_cherrypy.app.**Events**

Expose the Salt event bus

The event bus on the Salt master exposes a large variety of things, notably when executions are started on the master and also when minions ultimately return their results. This URL provides a real-time window into a running Salt infrastructure.

See also:

events

GET(*token=None*)

An HTTP stream of the Salt master event bus

This stream is formatted per the Server Sent Events (SSE) spec. Each event is formatted as JSON.

New in version 0.8.3.

Browser clients currently lack Cross-origin resource sharing (CORS) support for the `EventSource()` API. Cross-domain requests from a browser may instead pass the *X-Auth-Token* value as an URL parameter:

```
% curl -NsS localhost:8000/events/6d1b722e
```

GET /events

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

Example request:

```
% curl -NsS localhost:8000/events
```

```
GET /events HTTP/1.1
Host: localhost:8000
```

Example response:

```
HTTP/1.1 200 OK
Connection: keep-alive
Cache-Control: no-cache
Content-Type: text/event-stream;charset=utf-8

retry: 400
data: {'tag': '', 'data': {'minions': ['ms-4', 'ms-3', 'ms-2', 'ms-1', 'ms-0']}}

data: {'tag': '20130802115730568475', 'data': {'jid': '20130802115730568475', 'return': '
```

The event stream can be easily consumed via JavaScript:

```
# Note, you must be authenticated!
var source = new EventSource('/events');
source.onopen = function() { console.debug('opening') };
source.onerror = function(e) { console.debug('error!', e) };
source.onmessage = function(e) { console.debug(e.data) };
```

It is also possible to consume the stream via the shell.

Records are separated by blank lines; the `data:` and `tag:` prefixes will need to be removed manually before attempting to unserialize the JSON.

curl's `-N` flag turns off input buffering which is required to process the stream incrementally.

Here is a basic example of printing each event as it comes in:

```
% curl -NsS localhost:8000/events |\
        while IFS= read -r line ; do
            echo $line
        done
```

Here is an example of using awk to filter events based on tag:

```
% curl -NsS localhost:8000/events |\
        awk '
            BEGIN { RS=""; FS="\n" }
            $1 ~ /^tag: salt\/job\/[0-9]+\/new$/ { print $0 }
        '
tag: salt/job/20140112010149808995/new
```

```
        data: {"tag": "salt/job/20140112010149808995/new", "data": {"tgt_type": "glob", "jid": "2014
        tag: 20140112010149808995
        data: {"tag": "20140112010149808995", "data": {"fun_args": [], "jid": "20140112010149808995"
```

/ws

class saltapi.netapi.rest_cherrypy.app.**WebsocketEndpoint**

Open a WebSocket connection to Salt's event bus

The event bus on the Salt master exposes a large variety of things, notably when executions are started on the master and also when minions ultimately return their results. This URL provides a real-time window into a running Salt infrastructure. Uses websocket as the transport mechanism.

See also:

events

GET (*token=None*, ***kwargs*)

Return a websocket connection of Salt's event stream

New in version 0.8.6.

GET /ws/ (*token*)

Query Parameters

- **format_events** – The event stream will undergo server-side formatting if the format_events URL parameter is included in the request. This can be useful to avoid formatting on the client-side:

```
curl -NsS <...snip...> localhost:8000/ws?format_events
```

Request Headers

- *X-Auth-Token* – an authentication token from *Login*.

Status Codes

- 101 Switching Protocols – switching to the websockets protocol
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

Example request:

```
curl -NsS \
    -H 'X-Auth-Token: ffedf49d' \
    -H 'Host: localhost:8000' \
    -H 'Connection: Upgrade' \
    -H 'Upgrade: websocket' \
    -H 'Origin: http://localhost:8000' \
    -H 'Sec-WebSocket-Version: 13' \
    -H 'Sec-WebSocket-Key: '"$(echo -n $RANDOM | base64)" \
    localhost:8000/ws
```

```
GET /ws HTTP/1.1
Connection: Upgrade
Upgrade: websocket
Host: localhost:8000
Origin: http://localhost:8000
Sec-WebSocket-Version: 13
```

```
Sec-WebSocket-Key: s65VsgHigh7v/Jcf4nXHnA==
X-Auth-Token: ffedf49d
```

Example response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: mWZjBV9FCglzn1rIKJAxrTFlnJE=
Sec-WebSocket-Version: 13
```

An authentication token **may optionally** be passed as part of the URL for browsers that cannot be configured to send the authentication header or cookie:

```
curl -NsS <...snip...> localhost:8000/ws/ffedf49d
```

The event stream can be easily consumed via JavaScript:

```
// Note, you must be authenticated!
var source = new Websocket('ws://localhost:8000/ws/d0ce6c1a');
source.onerror = function(e) { console.debug('error!', e); };
source.onmessage = function(e) { console.debug(e.data); };

source.send('websocket client ready')

source.close();
```

Or via Python, using the Python module websocket-client for example.

```
# Note, you must be authenticated!

from websocket import create_connection

ws = create_connection('ws://localhost:8000/ws/d0ce6c1a')
ws.send('websocket client ready')

# Look at https://pypi.python.org/pypi/websocket-client/ for more examples.
while listening_to_events:
    print ws.recv()

ws.close()
```

Above examples show how to establish a websocket connection to Salt and activating real time updates from Salt's event stream by signaling `websocket client ready`.

/hook

class saltapi.netapi.rest_cherrypy.app.**Webhook**

A generic web hook entry point that fires an event on Salt's event bus

External services can POST data to this URL to trigger an event in Salt. For example, Amazon SNS, Jenkins-CI or Travis-CI, or GitHub web hooks.

Note: Be mindful of security

Salt's Reactor can run any code. A Reactor SLS that responds to a hook event is responsible for validating that the event came from a trusted source and contains valid data.

This is a generic interface and securing it is up to you!

This URL requires authentication however not all external services can be configured to authenticate. For this reason authentication can be selectively disabled for this URL. Follow best practices – always use SSL, pass a secret key, configure the firewall to only allow traffic from a known source, etc.

The event data is taken from the request body. The *Content-Type* header is respected for the payload.

The event tag is prefixed with `salt/netapi/hook` and the URL path is appended to the end. For example, a `POST` request sent to `/hook/mycompany/myapp/mydata` will produce a Salt event with the tag `salt/netapi/hook/mycompany/myapp/mydata`.

The following is an example `.travis.yml` file to send notifications to Salt of successful test runs:

```
language: python
script: python -m unittest tests
after_success:
    - 'curl -sS http://saltapi-url.example.com:8000/hook/travis/build/success -d branch="${TRAVI
```

See also:

events, reactor

POST (**args*, ***kwargs*)

Fire an event in Salt with a custom event tag and data

New in version 0.8.4.

POST /hook

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available
- 413 Request Entity Too Large – request body is too large

Example request:

```
% curl -sS localhost:8000/hook -d foo='Foo!' -d bar='Bar!'
```

```
POST /hook HTTP/1.1
Host: localhost:8000
Content-Length: 16
Content-Type: application/x-www-form-urlencoded

foo=Foo&bar=Bar!
```

Example response:

```
HTTP/1.1 200 OK
Content-Length: 14
Content-Type: application/json

{"success": true}
```

As a practical example, an internal continuous-integration build server could send an HTTP POST request to the URL `http://localhost:8000/hook/mycompany/build/success` which contains the result of a build and the SHA of the version that was built as JSON. That would then produce the following event in Salt that could be used to kick off a deployment via Salt's Reactor:

```
Event fired at Fri Feb 14 17:40:11 2014
*************************
Tag: salt/netapi/hook/mycompany/build/success
Data:
{'_stamp': '2014-02-14_17:40:11.440996',
    'headers': {
        'X-My-Secret-Key': 'F0fAgoQjIT@W',
        'Content-Length': '37',
        'Content-Type': 'application/json',
        'Host': 'localhost:8000',
        'Remote-Addr': '127.0.0.1'},
    'post': {'revision': 'aa22a3c4b2e7', 'result': True}}
```

Salt's Reactor could listen for the event:

```
reactor:
  - 'salt/netapi/hook/mycompany/build/*':
    - /srv/reactor/react_ci_builds.sls
```

And finally deploy the new build:

```
{% set secret_key = data.get('headers', {}).get('X-My-Secret-Key') %}
{% set build = data.get('post', {}) %}

{% if secret_key == 'F0fAgoQjIT@W' and build.result == True %}
deploy_my_app:
  cmd.state.sls:
    - tgt: 'application*'
    - arg:
      - myapp.deploy
    - kwarg:
        pillar:
          revision: {{ revision }}
{% endif %}
```

/stats

class saltapi.netapi.rest_cherrypy.app.**Stats**

Expose statistics on the running CherryPy server

GET()

Return a dump of statistics collected from the CherryPy server

GET /stats

Request Headers

- *X-Auth-Token* – a session token from *Login*.
- Accept – the desired response format.

Response Headers

- Content-Type – the format of the response body; depends on the *Accept* request header.

Status Codes

- 200 OK – success
- 401 Unauthorized – authentication required
- 406 Not Acceptable – requested Content-Type not available

rest_tornado

A REST API for Salt

depends

- tornado Python module

All Events Exposes `all` "real-time" events from Salt's event bus on a websocket connection. It should be noted that "Real-time" here means these events are made available to the server as soon as any salt related action (changes to minions, new jobs etc) happens. Clients are however assumed to be able to tolerate any network transport related latencies. Functionality provided by this endpoint is similar to the `/events` end point.

The event bus on the Salt master exposes a large variety of things, notably when executions are started on the master and also when minions ultimately return their results. This URL provides a real-time window into a running Salt infrastructure. Uses websocket as the transport mechanism.

Exposes GET method to return websocket connections. All requests should include an auth token. A way to obtain obtain authentication tokens is shown below.

```
% curl -si localhost:8000/login \
    -H "Accept: application/json" \
    -d username='salt' \
    -d password='salt' \
    -d eauth='pam'
```

Which results in the response

```
{
    "return": [{
        "perms": [".*", "@runner", "@wheel"],
        "start": 1400556492.277421,
        "token": "d0ce6c1a37e99dcc0374392f272fe19c0090cca7",
        "expire": 1400599692.277422,
        "user": "salt",
        "eauth": "pam"
    }]
}
```

In this example the `token` returned is `d0ce6c1a37e99dcc0374392f272fe19c0090cca7` and can be included in subsequent websocket requests (as part of the URL).

The event stream can be easily consumed via JavaScript:

```
// Note, you must be authenticated!

// Get the Websocket connection to Salt
var source = new Websocket('wss://localhost:8000/all_events/d0ce6c1a37e99dcc0374392f272fe19c0090cca7

// Get Salt's "real time" event stream.
source.onopen = function() { source.send('websocket client ready'); };

// Other handlers
source.onerror = function(e) { console.debug('error!', e); };

// e.data represents Salt's "real time" event data as serialized JSON.
```

```
source.onmessage = function(e) { console.debug(e.data); };

// Terminates websocket connection and Salt's "real time" event stream on the server.
source.close();
```

Or via Python, using the Python module websocket-client for example. Or the tornado client.

```
# Note, you must be authenticated!

from websocket import create_connection

# Get the Websocket connection to Salt
ws = create_connection('wss://localhost:8000/all_events/d0ce6c1a37e99dcc0374392f272fe19c0090cca7')

# Get Salt's "real time" event stream.
ws.send('websocket client ready')


# Simple listener to print results of Salt's "real time" event stream.
# Look at https://pypi.python.org/pypi/websocket-client/ for more examples.
while listening_to_events:
    print ws.recv()       #  Salt's "real time" event data as serialized JSON.

# Terminates websocket connection and Salt's "real time" event stream on the server.
ws.close()

# Please refer to https://github.com/liris/websocket-client/issues/81 when using a self signed cert
```

Above examples show how to establish a websocket connection to Salt and activating real time updates from Salt's event stream by signaling `websocket client ready`.

Formatted Events Exposes `formatted` "real-time" events from Salt's event bus on a websocket connection. It should be noted that "Real-time" here means these events are made available to the server as soon as any salt related action (changes to minions, new jobs etc) happens. Clients are however assumed to be able to tolerate any network transport related latencies. Functionality provided by this endpoint is similar to the `/events` end point.

The event bus on the Salt master exposes a large variety of things, notably when executions are started on the master and also when minions ultimately return their results. This URL provides a real-time window into a running Salt infrastructure. Uses websocket as the transport mechanism.

Formatted events parses the raw "real time" event stream and maintains a current view of the following:

- minions
- jobs

A change to the minions (such as addition, removal of keys or connection drops) or jobs is processed and clients are updated. Since we use salt's presence events to track minions, please enable `presence_events` and set a small value for the `loop_interval` in the salt master config file.

Exposes GET method to return websocket connections. All requests should include an auth token. A way to obtain obtain authentication tokens is shown below.

```
% curl -si localhost:8000/login \
    -H "Accept: application/json" \
    -d username='salt' \
    -d password='salt' \
    -d eauth='pam'
```

Which results in the response

```
{
    "return": [{
        "perms": [".*", "@runner", "@wheel"],
        "start": 1400556492.277421,
        "token": "d0ce6c1a37e99dcc0374392f272fe19c0090cca7",
        "expire": 1400599692.277422,
        "user": "salt",
        "eauth": "pam"
    }]
}
```

In this example the `token` returned is `d0ce6c1a37e99dcc0374392f272fe19c0090cca7` and can be included in subsequent websocket requests (as part of the URL).

The event stream can be easily consumed via JavaScript:

```
// Note, you must be authenticated!

// Get the Websocket connection to Salt
var source = new Websocket('wss://localhost:8000/formatted_events/d0ce6c1a37e99dcc0374392f272fe19c009

// Get Salt's "real time" event stream.
source.onopen = function() { source.send('websocket client ready'); };

// Other handlers
source.onerror = function(e) { console.debug('error!', e); };

// e.data represents Salt's "real time" event data as serialized JSON.
source.onmessage = function(e) { console.debug(e.data); };

// Terminates websocket connection and Salt's "real time" event stream on the server.
source.close();
```

Or via Python, using the Python module websocket-client for example. Or the tornado client.

```
# Note, you must be authenticated!

from websocket import create_connection

# Get the Websocket connection to Salt
ws = create_connection('wss://localhost:8000/formatted_events/d0ce6c1a37e99dcc0374392f272fe19c0090cca

# Get Salt's "real time" event stream.
ws.send('websocket client ready')


# Simple listener to print results of Salt's "real time" event stream.
# Look at https://pypi.python.org/pypi/websocket-client/ for more examples.
while listening_to_events:
    print ws.recv()       #  Salt's "real time" event data as serialized JSON.

# Terminates websocket connection and Salt's "real time" event stream on the server.
ws.close()
```

```
# Please refer to https://github.com/liris/websocket-client/issues/81 when using a self signed cert
```

Above examples show how to establish a websocket connection to Salt and activating real time updates from Salt's event stream by signaling `websocket client ready`.

Example responses `Minion information` is a dictionary keyed by each connected minion's `id` (`mid`), grains information for each minion is also included.

Minion information is sent in response to the following minion events:

- **connection drops**
  - requires running `manage.present` periodically every `loop_interval` seconds
- minion addition
- minon removal

```
# Not all grains are shown
data: {
    "minions": {
        "minion1": {
            "id": "minion1",
            "grains": {
                "kernel": "Darwin",
                "domain": "local",
                "zmqversion": "4.0.3",
                "kernelrelease": "13.2.0"
            }
        }
    }
}
```

`Job information` is also tracked and delivered.

Job information is also a dictionary in which each job's information is keyed by salt's `jid`.

```
data: {
    "jobs": {
        "20140609153646699137": {
            "tgt_type": "glob",
            "jid": "20140609153646699137",
            "tgt": "*",
            "start_time": "2014-06-09T15:36:46.700315",
            "state": "complete",
            "fun": "test.ping",
            "minions": {
                "minion1": {
                    "return": true,
                    "retcode": 0,
                    "success": true
                }
            }
        }
    }
}
```

Setup

In order to run rest_tornado with the salt-master add the following to your salt master config file.

```
rest_tornado:
    # can be any port
    port: 8000
    ssl_crt: /etc/pki/api/certs/server.crt
    # no need to specify ssl_key if cert and key
    # are in one single file
    ssl_key: /etc/pki/api/certs/server.key
    debug: False
    disable_ssl: False
```

rest_wsgi

A minimalist REST API for Salt

This `rest_wsgi` module provides a no-frills REST interface to a running Salt master. There are no dependencies.

Please read this introductory section in entirety before deploying this module.

configuration All authentication is done through Salt's external auth system. Be sure that it is enabled and the user you are authenticating as has permissions for all the functions you will be running.

The configuration options for this module resides in the Salt master config file. All available options are detailed below.

port Required

The port for the webserver to listen on.

Example configuration:

```
rest_wsgi:
  port: 8001
```

This API is not very "RESTful"; please note the following:

- All requests must be sent to the root URL (`/`).
- All requests must be sent as a POST request with JSON content in the request body.
- All responses are in JSON.

See also:

rest_cherrypy

The *rest_cherrypy* module is more full-featured, production-ready, and has builtin security features.

Deployment

The `rest_wsgi` netapi module is a standard Python WSGI app. It can be deployed one of two ways.

`salt-api` using a development-only server If run directly via salt-api it uses the wsgiref.simple_server() that ships in the Python standard library. This is a single-threaded server that is intended for testing and development. This server does **not** use encryption; please note that raw Salt authentication credentials must be sent with every HTTP request.

Running this module via salt-api is not recommended for most use!

Using a WSGI-compliant web server This module may be run via any WSGI-compliant production server such as Apache with mod_wsgi or Nginx with FastCGI.

It is highly recommended that this app be used with a server that supports HTTPS encryption since raw Salt authentication credentials must be sent with every request. Any apps that access Salt through this interface will need to manually manage authentication credentials (either username and password or a Salt token). Tread carefully.

Usage examples

`POST /`

Example request for a basic `test.ping`:

```
% curl -sS -i \
        -H 'Content-Type: application/json' \
        -d '[{"eauth":"pam","username":"saltdev","password":"saltdev","client":"local","tgt":"*"
```

Example response:

```
HTTP/1.0 200 OK
Content-Length: 89
Content-Type: application/json

{"return": [{"ms--4": true, "ms--3": true, "ms--2": true, "ms--1": true, "ms--0": true}]}
```

Example request for an asyncronous `test.ping`:

```
% curl -sS -i \
        -H 'Content-Type: application/json' \
        -d '[{"eauth":"pam","username":"saltdev","password":"saltdev","client":"local_async","tg
```

Example response:

```
HTTP/1.0 200 OK
Content-Length: 103
Content-Type: application/json

{"return": [{"jid": "20130412192112593739", "minions": ["ms--4", "ms--3", "ms--2", "ms--1", "ms-
```

Example request for looking up a job ID:

```
% curl -sS -i \
        -H 'Content-Type: application/json' \
        -d '[{"eauth":"pam","username":"saltdev","password":"saltdev","client":"runner","fun":"j
```

Example response:

```
HTTP/1.0 200 OK
Content-Length: 89
Content-Type: application/json

{"return": [{"ms--4": true, "ms--3": true, "ms--2": true, "ms--1": true, "ms--0": true}]}
```

form lowstate A list of *lowstate* data appropriate for the client interface you are calling.

status 200 success

status 401 authentication required

3.2 `netapi` developer reference

3.2.1 Introduction to netapi modules

netapi modules generally bind to a port and start a service. They are purposefully open-ended. There could be multiple netapi modules that provide a REST interface, a module that provides an XMPP interface, or Websockets, or XMLRPC.

netapi modules are enabled by adding configuration to your master config file. Check the docs for each module to see external requirements and configuration settings.

Communication with Salt and Salt satellite projects is done by passing a list of lowstate dictionaries to a client interface. A list of available client interfaces is below. The lowstate dictionary items map to keyword arguments on the client interface.

See also:

python-api

Client interfaces

class saltapi.**APIClient**(*opts*)

Provide a uniform method of accessing the various client interfaces in Salt in the form of low-data data structures. For example:

```
>>> client = APIClient(__opts__)
>>> lowstate = {'client': 'local', 'tgt': '*', 'fun': 'test.ping', 'arg': ''}
>>> client.run(lowstate)
```

local(**args*, ***kwargs*)

Run execution modules syncronously

Wraps `salt.client.LocalClient.cmd()`.

Returns Returns the result from the execution module

local_async(**args*, ***kwargs*)

Run execution modules asyncronously

Wraps `salt.client.LocalClient.run_job()`.

Returns job ID

local_batch(**args*, ***kwargs*)

Run execution modules against batches of minions

New in version 0.8.4.

Wraps `salt.client.LocalClient.cmd_batch()`

Returns Returns the result from the exeuction module for each batch of returns

runner (*fun*, ***kwargs*)
Run *runner modules <all-salt.runners>*

Wraps `salt.runner.RunnerClient.low()`.

Returns Returns the result from the runner module

wheel (*fun*, ***kwargs*)
Run wheel modules

Wraps `salt.wheel.WheelClient.master_call()`.

Returns Returns the result from the wheel module

3.2.2 Writing netapi modules

`netapi` modules, put simply, bind a port and start a service. They are purposefully open-ended and can be used to present a variety of external interfaces to Salt, and even present multiple interfaces at once.

See also:

The full list of netapi modules

Configuration

All `netapi` configuration is done in the Salt master config and takes a form similar to the following:

```
rest_cherrypy:
  port: 8000
  debug: True
  ssl_crt: /etc/pki/tls/certs/localhost.crt
  ssl_key: /etc/pki/tls/certs/localhost.key
```

The `__virtual__` function

Like all module types in Salt, `netapi` modules go through Salt's loader interface to determine if they should be loaded into memory and then executed.

The `__virtual__` function in the module makes this determination and should return `False` or a string that will serve as the name of the module. If the module raises an `ImportError` or any other errors, it will not be loaded.

The `start` function

The `start()` function will be called for each `netapi` module that is loaded. This function should contain the server loop that actually starts the service. This is started in a multiprocess.

Inline documentation

As with the rest of Salt, it is a best-practice to include liberal inline documentation in the form of a module docstring and docstrings on any classes, methods, and functions in your `netapi` module.

Loader "magic" methods

The loader makes the `__opts__` data structure available to any function in a `netapi` module.

CHAPTER 4

Releases

4.1 Release notes

4.1.1 salt-api 0.5.0

salt-api is gearing up for the initial public release with 0.5.0. Although this release ships with working basic functionality it is awaiting the authentication backend that will be introduced in Salt 0.10.4 before it can be considered ready for testing at large.

REST API

This release presents the flagship netapi module which provides a RESTful interface to a running Salt system. It allows for viewing minions, runners, and jobs as well as running execution modules and runners of a running Salt system through a REST API that returns JSON.

Participation

salt-api is just getting off the ground so feedback, questions, and ideas are critical as we solidify how this project fits into the overall Salt infrastructure management stack. Please get involved by filing issues on GitHub, discussing on the mailing list, and chatting in `#salt` on Freenode.

4.1.2 salt-api 0.6.0

salt-api inches closer to prime-time with 0.6.0. This release adds the beginnings of a universal interface for accessing Salt components via the tried and true method of passing low-data to functions (a core component of Salt's remote execution and state management).

Low-data interface

A new view accepts :http:post: requests at the root URL that accepts raw low-data as :http:post: data and passes that low-data along to a client interface in Salt. Currently only LocalClient and RunnerClient interfaces have been implemented in Salt with more coming in the next Salt release.

External authentication

Raw low-data can contain authentication credentials that make use of Salt's new `external_auth` system.

The following is a proof-of-concept of a working eauth call. (It bears repeating this is a pre-alpha release and this should not be used by anyone for anything real.)

```
% curl -si localhost:8000 \
    -d client=local \
    -d tgt='*' \
    -d fun='test.ping' \
    -d arg \
    -d eauth=pam \
    -d username=saltdev \
    -d password=saltdev
```

Participation

salt-api is just getting off the ground so feedback, questions, and ideas are critical as we solidify how this project fits into the overall Salt infrastructure management stack. Please get involved by filing issues on GitHub, discussing on the mailing list, and chatting in `#salt-devel` on Freenode.

4.1.3 salt-api 0.7.0

salt-api is ready for alpha-testing in the real world. This release solidifies how **salt-api** will communicate with the larger Salt ecosystem. In addition authentication and encryption (via SSL) have been added.

The first netapi module was a proof of concept written in Flask. It was quite useful to be able to quickly hammer out a URL structure and solidify on an interface for programmatically calling out to Salt components. As of this release that module has been deprecated and removed in favor of a netapi module written in CherryPy. CherryPy affords tremendous flexibility when composing a REST interface and will present a stable platform for building out a very adaptable and featureful REST API—also we're using the excellent and fast CherryPy webserver for securely serving the API.

Low-data interface

The last release introduced a proof-of-concept for how the various Salt components will communicate with each other. This is done by passing a data structure to a client interface. This release expands on that. There are currently three client interfaces in Salt.

See also:

Introduction to netapi modules

Encryption and authentication

Encryption has been added via SSL. You can supply an existing certificate or generate a self-signed certificate through Salt's `tls` module.

Authentication is performed through Salt's incredibly flexible external auth system and is maintained when accessing the API via session tokens.

Participation

salt-api is just getting off the ground so feedback, questions, and ideas are critical as we solidify how this project fits into the overall Salt infrastructure management stack. Please get involved by filing issues on GitHub, discussing on the mailing list, and chatting in `#salt-devel` on Freenode.

4.1.4 salt-api 0.7.5

This release is a mostly a minor release to pave a better path for **salt-ui** though there are some small feature additions and bugfixes.

Changes

- Convenience URLs `/minions` and `/jobs` have been added as well as a async client wrapper. This starts a job and immediately returns the job ID, allowing you to fetch the result of that job at a later time.
- The return format will now default to JSON if no specific format is requested.
- A new setting `static` has been added that will serve any static media from the directory specified. In addition if an `index.html` file is found in that directory and the `Accept` header in the request prefer HTML that file will be served.
- All HTML, including the login form, has been removed from **salt-api** and moved into the **salt-ui** project.
- Sessions now live as long as the Salt token.

Participation

salt-api is just getting off the ground so feedback, questions, and ideas are critical as we solidify how this project fits into the overall Salt infrastructure management stack. Please get involved by filing issues on GitHub, discussing on the mailing list, and chatting in `#salt-devel` on Freenode.

4.1.5 salt-api 0.8.0

We are happy to announce the release of **salt-api** 0.8.0.

This release encompasses bugfixes and new features for the *rest_cherrypy* netapi module that provides a RESTful interface for a running Salt system.

Note: Requires Salt 0.13

Changes

In addition to the usual documentation improvements and bug fixes this release introduces the following changes and additions.

Please note the backward incompatible change detailed below.

RPM packaging

Thanks to Andrew Niemantsvedriet (@kaptk2) **salt-api** is now available in Fedora package repositories as well as RHEL compatible systems via EPEL.

- http://dl.fedoraproject.org/pub/epel/5/i386/repoview/salt-api.html
- http://dl.fedoraproject.org/pub/epel/5/x86_64/repoview/salt-api.html
- http://dl.fedoraproject.org/pub/epel/6/i386/repoview/salt-api.html
- http://dl.fedoraproject.org/pub/epel/6/x86_64/repoview/salt-api.html

Thanks also to Clint Savage (@herlo) and Thomas Spura (@tomspur) for helping with that process.

Ubuntu PPA packaging

Thanks to Sean Channel (@seanchannel, pentabular) **salt-api** is available as a PPA on the SaltStack LaunchPad team.

https://launchpad.net/~saltstack/+archive/salt

Authentication information on login

> **Warning:** Backward incompatible change
> The */login* URL no longer responds with a 302 redirect for success.
> Although this is behavior is common in the browser world it is not useful from an API so we have changed it to return a 200 response in this release.
> We take backward compatibility very seriously and we apologize for the inconvenience. In this case we felt the previous behavior was limiting. Changes such as this will be rare.

New in this release is displaying information about the current session and the current user. For example:

```
% curl -sS localhost:8000/login \
    -H 'Accept: application/x-yaml'
    -d username='saltdev'
    -d password='saltdev'
    -d eauth='pam'

return:
- eauth: pam
  expire: 1365508324.359403
  perms:
  - '@wheel'
  - grains.*
  - state.*
  - status.*
  - sys.*
  - test.*
  start: 1365465124.359402
  token: caa7aa2b9dbc4a8adb6d2e19c3e52be68995ef4b
  user: saltdev
```

Bypass session handling

A convenience URL has been added (*/run*) to bypass the normal session-handling process.

The REST interface uses the concept of "lowstate" data to specify what function should be executed in Salt (plus where that function is and any arguments to the function). This is a thin wrapper around Salt's various "client" interfaces, for example Salt's LocalClient() which can accept authentication credentials directly.

Authentication with the REST API typically goes through the login URL and a session is generated that is tied to a Salt external_auth token. That token is then automatically added to the lowstate for subsequent requests that match the current session.

It is sometimes useful to handle authentication or token management manually from another program or script. For example:

```
curl -sS localhost:8000/run \
    -d client='local' \
    -d tgt='*' \
    -d fun='test.ping' \
    -d eauth='pam' \
    -d username='saltdev' \
    -d password='saltdev'
```

It is a Bad Idea (TM) to do this unless you have a very good reason and a well thought out security model.

Logout

An URL has been added (*/logout*) that will cause the client-side to expire the session cookie and the server-side session to be invalidated.

Running the REST interface via any WSGI-compliant server

The *rest_cherrypy* netapi module is a regular WSGI application written using the CherryPy framework. It was written with the intent of also running from any WSGI-compliant server such as Apache and mod_wsgi, Gunicorn, uWSGI, Nginx and FastCGI, etc.

The WSGI application entry point has been factored out into a stand-alone file in this release suitable for calling from an external server. **salt-api** does not need to be running in this scenario.

For example, an Apache virtual host configuration:

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias *.example.com

    ServerAdmin webmaster@example.com

    LogLevel warn
    ErrorLog /var/www/example.com/logs/error.log
    CustomLog /var/www/example.com/logs/access.log combined

    DocumentRoot /var/www/example.com/htdocs

    WSGIScriptAlias / /path/to/saltapi/netapi/rest_cherrypy/wsgi.py
</VirtualHost>
```

Participation

Please get involved by filing issues on GitHub, discussing on the mailing list, and chatting in `#salt-devel` on Freenode.

4.1.6 salt-api 0.8.2

`salt-api` 0.8.2 is largely a bugfix release that fixes a compatibility issue with changes in Salt 0.15.9.

Note: Requires Salt 0.15.9 or greater

The following changes have been made to the *`rest_cherrypy`* netapi module that provides a RESTful interface for a running Salt system:

- Fixed issue #87 which caused the Salt master's PID file to be overwritten.
- Fixed an inconsistency with the return format for the `/minions` convenience URL.

  Warning: This is a backward incompatible change.

- Added a dedicated URL for serving an HTML app
- Added dedicated URL for serving static media

4.1.7 salt-api 0.8.3

`salt-api` 0.8.3 is a small release largely concerning changes and fixes to the *`rest_cherrypy`* netapi module.

This release will likely be the final salt-api release as a separate project. The Salt team has begun the process of merging this project directly in to the main Salt project. What this means for end users is only that there will be one fewer package to install. Salt itself will ship with the current `netapi` modules and the API and configuration will remain otherwise unchanged.

The reasoning behind merging the two projects is simply to lower the barrier to entry. Having a separate project was useful for experimentation and exploration but there was no technical reason for the separation – salt-api uses the same flexible module system that Salt uses and those modules will simply be moved into Salt.

Going forward, Salt will ship with the same REST interface that salt-api currently provides. This will have the side benefit of not having to coordinate incompatible Salt and salt-api releases.

`rest_cherrypy` changes

An HTTP stream of Salt's event bus has been added. This stream conforms to the SSE (Server Sent Events) spec and is easily consumed via JavaScript clients. This HTTP stream allows a real-time window into a running Salt system. A client watching the stream can see as soon as individual minions return data for a job, authentication events, and any other events that go through the Salt master.

A new configuration option to only allow access to whitelisted IP addresses. Of course, IP addresses can be easily spoofed so this feature should be thought of as a usability addition and not used for security purposes.

An option to disable SSL has been added. Previously SSL could only be disabled while running the HTTP server with debugging options enabled. Now each item can be enabled or disabled independently of the other.

In addition, there has been several bug fixes, packaging fixes, and minor code simplification.

4.1.8 salt-api 0.8.4

salt-api 0.8.4 sees a number of new features and feature enhancements in the *rest_cherrypy* netapi module.

Work to merge **salt-api** into the main Salt distribution continues and it is likely to be included in Salt's Helium release.

rest_cherrypy changes

Web hooks

This release adds a *new URL /hook* that allows salt-api to serve as a generic web hook interface for Salt. POST requests to the URL trigger events on Salt's event bus.

External services like Amazon SNS, Travis CI, GitHub, etc can easily send signals through Salt's Reactor.

The following HTTP call will trigger the following Salt event.

```
% curl -sS http://localhost:8000/hook/some/tag \
    -d some='Data!'
```

Event tag: `salt/netapi/hook/some/tag`. Event data:

```
{
  "_stamp": "2014-04-04T12:14:54.389614",
  "post": {
    "some": "Data!"
  },
  "headers": {
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "localhost:8000",
    "User-Agent": "curl/7.32.0",
    "Accept": "*/*",
    "Content-Length": "10",
    "Remote-Addr": "127.0.0.1"
  }
}
```

Batch mode

The *local_batch()* client exposes Salt's batch mode for executing commands on incremental subsets of minions.

Tests!

We have added the necessary framework for testing the rest_cherrypy module and this release includes a number of both unit and integration tests. The suite can be run with the following command:

```
python -m unittest discover -v
```

CherryPy server stats and configuration

A number of settings have been added to better configure the performance of the CherryPy web server. In addition, a *new URL /stats* has been added to expose metrics on the health of the CherryPy web server.

Improvements for running with external WSGI servers

Running the `rest_cherrypy` module via a WSGI-capable server such as Apache or Nginx can be tricky since the user the server is running as must have permission to access the running Salt system. This release eases some of those restrictions by accessing Salt's key interface through the external auth system. Read access to the Salt configuration is required for the user the server is running as and everything else should go through external auth.

More information in the jobs URLs

The output for the `/jobs/` has been augmented with more information about the job such as which minions are expected to return for that job. This same output will be added to the other salt-api URLs in the next release.

Improvements to the Server Sent Events stream

Event tags have been added to `the HTTP event stream` as SSE tags which allows JavaScript or other consumers to more easily match on certain tags without having to inspect the whole event.

CHAPTER 5

Reference

- genindex
- modindex
- search
- glossary

/

GET /, 11
POST /, 28

/events

GET /events, 18

/hook

POST /hook, 21

/jobs

GET /jobs/(jid), 15

/login

GET /login, 12
POST /login, 13

/minions

GET /minions/(mid), 14
POST /minions, 14

/run

POST /run, 16

/stats

GET /stats, 22

/ws

GET /ws/(token), 19

n

saltapi.netapi.rest_cherrypy.app, 7
saltapi.netapi.rest_cherrypy.wsgi, 10
saltapi.netapi.rest_tornado.saltnado, 23
saltapi.netapi.rest_wsgi, 27

A

APIClient (class in saltapi), 29

E

Events (class in saltapi.netapi.rest_cherrypy.app), 17

G

GET() (saltapi.netapi.rest_cherrypy.app.Events method), 17
GET() (saltapi.netapi.rest_cherrypy.app.Jobs method), 15
GET() (saltapi.netapi.rest_cherrypy.app.Login method), 12
GET() (saltapi.netapi.rest_cherrypy.app.LowDataAdapter method), 11
GET() (saltapi.netapi.rest_cherrypy.app.Minions method), 14
GET() (saltapi.netapi.rest_cherrypy.app.Stats method), 22
GET() (saltapi.netapi.rest_cherrypy.app.WebsocketEndpoint method), 19

J

Jobs (class in saltapi.netapi.rest_cherrypy.app), 15

L

local() (saltapi.APIClient method), 29
local_async() (saltapi.APIClient method), 29
local_batch() (saltapi.APIClient method), 29
Login (class in saltapi.netapi.rest_cherrypy.app), 12
Logout (class in saltapi.netapi.rest_cherrypy.app), 14
LowDataAdapter (class in saltapi.netapi.rest_cherrypy.app), 11
lowstate, **9**

M

Minions (class in saltapi.netapi.rest_cherrypy.app), 14

P

POST (saltapi.netapi.rest_cherrypy.app.LowDataAdapter attribute), 11
POST() (saltapi.netapi.rest_cherrypy.app.Login method), 13
POST() (saltapi.netapi.rest_cherrypy.app.Logout method), 14
POST() (saltapi.netapi.rest_cherrypy.app.Minions method), 14
POST() (saltapi.netapi.rest_cherrypy.app.Run method), 16
POST() (saltapi.netapi.rest_cherrypy.app.Webhook method), 21

R

Run (class in saltapi.netapi.rest_cherrypy.app), 16
runner() (saltapi.APIClient method), 29

S

saltapi.netapi.rest_cherrypy.app (module), 7
saltapi.netapi.rest_cherrypy.wsgi (module), 10
saltapi.netapi.rest_tornado.saltnado (module), 23
saltapi.netapi.rest_wsgi (module), 27
Stats (class in saltapi.netapi.rest_cherrypy.app), 22

W

Webhook (class in saltapi.netapi.rest_cherrypy.app), 20
WebsocketEndpoint (class in saltapi.netapi.rest_cherrypy.app), 19
wheel() (saltapi.APIClient method), 30