autopush Documentation

Release 1.33.0

Mozilla

Jul 21, 2017

Contents

Mozilla Push server and Push Endpoint utilizing PyPy, twisted, and DynamoDB.

This is the third generation of Push server built in Mozilla Services, first to handle Push for FirefoxOS clients, then extended for push notifications for Firefox (via the W3C Push spec.)

For how to read and respond to **autopush error codes**, see *Errors*.

For an overview of the Mozilla Push Service and where autopush fits in, see the Mozilla Push Service architecture diagram. This push service uses websockets to talk to Firefox, with a Push endpoint that implements the *WebPush* standard for its *HTTP Endpoints for Notifications* API.

CHAPTER 1

Autopush APIs

For developers writing mobile applications in Mozilla, or web developers using Push on the web with Firefox.

HTTP Endpoints for Notifications

Autopush exposes three HTTP endpoints:

/[sw]push/...

This is tied to one of the Endpoint Handlers, either */wpush/...* for `WebPushHandler`, or */spush/...* for `SimplePushHandler`. (Note, *SimplePushHandler* is obsolete and will be removed without notification in a future update.) This endpoint is returned by the Push registration process and is used by the *AppServer* to send Push alerts to the Application. See *Send Notification*.

/m/...

This is tied to `MessageHandler`. This endpoint allows a message that has not yet been delivered to be deleted. See *Cancel Notification*.

/v1/.../.../registration/...

This is tied to the *Registration* Handlers. This endpoint is used by apps that wish to use *bridging* protocols to register new channels. See *Push Service Bridge HTTP Interface*.

—

Push Service HTTP API

The following section describes how remote servers can send Push Notifications to apps running on remote User Agents.

Lexicon

{UAID} The Push User Agent Registration ID

Push assigns each remote recipient a unique identifier. {UAID}s are UUIDs in lower case, undashed format. (e.g. '01234567abcdabcdabcd01234567abcd') This value is assigned during **Registration**

{CHID} The *Channel* Subscription ID

Push assigns a unique identifier for each subscription for a given {UAID}. Like {UAID}s, {CHID}s are UUIDs, but in lower case, dashed format(e.g. '01234567-abcd-abcd-abcd-0123456789ab'). The User Agent usually creates this value and passes it as part of the **Channel Subscription**. If no value is supplied, the server will create and return one.

{message-id} The unique Message ID

Push assigns each message for a given Channel Subscription a unique identifier. This value is assigned during **Send Notification**.

Response

The responses will be JSON formatted objects. In addition, API calls will return valid HTTP error codes (see *Error Codes* sub-section for descriptions of specific errors).

For non-success responses, an extended error code object will be returned with the following format:

```
{
    "code": 404,  // matches the HTTP status code
    "errno": 103, // stable application-level error number
    "error": "Not Found", // string representation of the status
    "message": "No message found" // optional additional error information
}
```

Error Codes

Autopush uses error codes based on HTTP response codes. An error response will contain a JSON body including an additional error information (see *Response*).

Unless otherwise specified, all calls return one the following error statuses:

- 20x - **Success** - The message was accepted for transmission to the client. Please note that the message may still be rejected by the User Agent if there is an error with the message's encryption.
- 301 - **Moved + 'Location:'** if *{client_token}* is invalid (Bridge API Only) - Bridged services (ones that run over third party services like GCM and APNS), may require a new URL be used. Please stop using the old URL immediately and instead use the new URL provided.
- 400 - **Bad Parameters** – One or more of the parameters specified is invalid. See the following sub-errors indicated by *errno*
  - errno 101 - Missing necessary crypto keys - One or more required crypto key elements are missing from this transaction. Refer to the appropriate specification for the requested content-type.
  - errno 108 - Router type is invalid - The URL contains an invalid router type, which may be from URL corruption or an unsupported bridge. Refer to *Push Service Bridge HTTP Interface*.
  - errno 110 - Invalid crypto keys specified - One or more of the crytpo key elements are invalid. Refer to the appropriate specification for the requested content-type.

  - errno 111 - Missing Required Header - A required crypto element header is missing. Refer to the appropriate specification for the requested content-type.
    * Missing TTL Header - Include the Time To Live header (IETF WebPush protocol §6.2)
    * Missing Crypto Headers - Include the appropriate encryption headers (WebPush Encryption §3.2 and WebPush VAPID §4)
  - errno 112 - Invalid TTL header value - The Time To Live "TTL" header contains an invalid or unreadable value. Please change to a number of seconds that this message should live, between 0 (message should be dropped immediately if user is unavailable) and 2592000 (hold for delivery within the next approximately 30 days).
  - errno 113 - Invalid Topic header value - The Topic header contains an invalid or unreadable value. Please use only ASCII alphanumeric values [A-Za-z0-9] and a maximum length of 32 bytes..
- 401 - **Bad Authorization** - *Authorization* header is invalid or missing. See the VAPID specification.
  - errno 109 - Invalid authentication
- 404 - **Endpoint Not Found** - The URL specified is invalid and should not be used again.
  - errno 102 - Invalid URL endpoint
- 410 - **Endpoint Not Valid** - The URL specified is no longer valid and should no longer be used. A User has become permanently unavailable at this URL.
  - errno 103 - Expired URL endpoint
  - errno 105 - Endpoint became unavailable during request
  - errno 106 - Invalid subscription
- 413 - **Payload too large** - The body of the message to send is too large. The max data that can be sent is 4028 characters. Please reduce the size of the message.
  - errno 104 - Data payload too large
- 500 - **Unknown server error** - An internal error occurred within the Push Server.
  - errno 999 - Unknown error
- 503 - **Server temporarily unavaliable.** - The Push Service is currently unavailable. See the error number "errno" value to see if retries are available.
  - errno 201 - Use exponential back-off for retries
  - errno 202 - Immediate retry ok

Calls

Send Notification

Send a notification to the given endpoint identified by its *push_endpoint*. Please note, the Push endpoint URL (which is what is used to send notifications) should be considered "opaque". We reserve the right to change any portion of the Push URL in future provisioned URLs.

The *Topic* HTTP header allows new messages to replace previously sent, unreceived subscription updates. See *Message Topics*.

Call:

POST {push_endpoint}

If the client is using webpush style data delivery, then the body in its entirety will be regarded as the data payload for the message per the WebPush spec.

Note: Some bridged connections require data transcription and may limit the length of data that can be sent. For instance, using a GCM/FCM bridge will require that the data be converted to base64. This means that data may be limited to only 2744 bytes instead of the normal 4096 bytes.

Reply:

```
{"message-id": {message-id}}
```

Return Codes:

statuscode 404 Push subscription is invalid.

statuscode 202 Message stored for delivery to client at a later time.

statuscode 200 Message delivered to node client is connected to.

Message Topics

Message topics allow newer message content to replace previously sent, unread messages. This prevents the UA from displaying multiple messages upon reconnect. A blog post provides an example of how to use Topics, but a summary is provided here.

To specify a Topic, include a *Topic* HTTP header along with your *Send Notification*. The topic can be any 32 byte alpha-numeric string (including "_" and "-").

Example topics might be *MailMessages*, *Current_Score*, or *20170814-1400_Meeting_Reminder*

For example:

```
curl -X POST \
    https://push.services.mozilla.com/wpush/abc123... \
    -H "TTL: 86400" \
    -H "Topic: new_mail" \
    -H "Authorization: Vapid AbCd..." \
    ...
```

Would create or replace a message that is valid for the next 24 hours that has the topic of *new_mail*. The body of this might contain the number of unread messages. If a new message arrives, the Application Server could send a second message with a body containing a revised message count.

Later, when the User reconnects, she will only see a single notification containing the latest notification, with the most recent new mail message count.

Cancel Notification

Delete the message given the *message_id*.

Call:

DELETE /m/{message_id}

Parameters:

None

Reply:

```
{}
```

Return Codes:

See *Error Codes*.

—

Push Service Bridge HTTP Interface

Push allows for remote devices to perform some functions using an HTTP interface. This is mostly used by devices that are bridging via an external protocol like GCM/FCM or APNs. All message bodies must be UTF-8 encoded.

API methods requiring Authorization must provide the Authorization header containing the registration secret. The registration secret is returned as "secret" in the registration response.

Lexicon

For the following call definitions:

{type} The bridge type.

Allowed bridges are *gcm* (Google Cloud Messaging), *fcm* (Firebase Cloud Messaging), and *apns* (Apple Push Notification system)

{app_id} The bridge specific application identifier

Each bridge may require a unique token that addresses the remote application For GCM/FCM, this is the *SenderID* (or 'project number') and is pre-negotiated outside of the push service. You can find this number using the Google developer console. For APNS, this value is the "platform" or "channel" of development (e.g. "firefox", "beta", "gecko", etc.) For our examples, we will use a client token of "33clienttoken33".

{instance_id} The bridge specific private identifier token

Each bridge requires a unique token that addresses the application on a given user's device. This is the "Registration Token" for GCM/FCM or "Device Token" for APNS. This is usually the product of the application registering the {instance_id} with the native bridge via the user agent. For our examples, we will use an instance ID of "11-instance-id-11".

{secret} The registration secret from the Registration call.

Most calls to the HTTP interface require a Authorization header. The Authorization header is a simple bearer token, which has been provided by the **Registration** call and is preceded by the scheme name "Bearer". For our examples, we will use a registration secret of "00secret00".

An example of the Authorization header would be:

```
Authorization: Bearer 00secret00
```

Calls

Registration

Request a new UAID registration, Channel ID, and optionally set a bridge type and 3rd party bridge instance ID token for this connection. (See `NewRegistrationHandler`)

Call:

`POST /v1/{type}/{app_id}/registration`

This call requires no Authorization header.

Parameters:

{"token":{instance_id}}

> **Note:** If additional information is required for the bridge, it may be included in the parameters as JSON elements. Currently, no additional information is required.

Reply:

```
`{"uaid": {UAID}, "secret": {secret},
"endpoint": "https://updates-push...", "channelID": {CHID}}`
```

example:

```
> POST /v1/fcm/33clienttoken33/registration
>
> {"token": "11-instance-id-11"}
```

```
< {"uaid": "01234567-0000-1111-2222-0123456789ab",
< "secret": "00secret00",
< "endpoint": "https://updates-push.services.mozaws.net/push/...",
< "channelID": "00000000-0000-1111-2222-0123456789ab"}
```

Return Codes:

See *Error Codes*.

Token updates

Update the current bridge token value. Note, this is a ***PUT*** call, since we are updating existing information. (See *UaidRegistrationHandler*)

Call:

`PUT /v1/{type}/{app_id}/registration/{uaid}`

```
Authorization: Bearer {secret}
```

Parameters:

{"token": {instance_id}}

> **Note:** If additional information is required for the bridge, it may be included in the parameters as JSON elements. Currently, no additional information is required.

Reply:

```
{}
```

example:

```
> PUT /v1/fcm/33clienttoken33/registration/abcdef012345
> Authorization: Bearer 00secret00
>
> {"token": "22-instance-id-22"}
```

```
< {}
```

Return Codes:

See *Error Codes*.

Channel Subscription

Acquire a new ChannelID for a given UAID. (See *SubRegistrationHandler*)

Call:

POST /v1/{type}/{app_id}/registration/{uaid}/subscription

```
Authorization: Bearer {secret}
```

Parameters:

{}

Reply:

```
{"channelID": {CHID}, "endpoint": "https://updates-push..."}
```

example:

```
> POST /v1/fcm/33clienttoken33/registration/abcdef012345/subscription
> Authorization: Bearer 00secret00
>
> {}
```

```
< {"channelID": "01234567-0000-1111-2222-0123456789ab",
< "endpoint": "https://updates-push.services.mozaws.net/push/..."}
```

Return Codes:

See *Error Codes*.

Unregister UAID (and all associated ChannelID subscriptions)

Indicate that the UAID, and by extension all associated subscriptions, is no longer valid. (See *UaidRegistrationHandler*)

Call:

DELETE /v1/{type}/{app_id}/registration/{uaid}

```
Authorization: Bearer {secret}
```

Parameters:

{}

Reply:

```
{}
```

Return Codes:

See *Error Codes*.

Unsubscribe Channel

Remove a given ChannelID subscription from a UAID. (See: *ChannelRegistrationHandler*)

Call:

DELETE /v1/{type}/{app_id}/registration/{UAID}/subscription/{CHID}

```
Authorization: Bearer {secret}
```

Parameters:

{}

Reply:

```
{}
```

Return Codes:

See *Error Codes*.

Get Known Channels for a UAID

Fetch the known ChannelIDs for a given bridged endpoint. This is useful to check link status. If no channelIDs are present for a given UAID, an empty set of channelIDs will be returned. (See: *UaidRegistrationHandler*)

Call:

GET /v1/{type}/{app_id}/registration/{UAID}/
Authorization: Bearer {secret}

Parameters:

{}

Reply:

```
{"uaid": {UAID}, "channelIDs": [{ChannelID}, ...]}
```

example:

```
> GET /v1/gcm/33clienttoken33/registration/abcdef012345/
> Authorization: Bearer 00secret00
>
> {}
```

```
< {"uaid": "abcdef012345",
< "channelIDS": ["01234567-0000-1111-2222-0123456789ab", "76543210-0000-1111-2222-
↪0123456789ab"]}
```

Return Codes:

See *Error Codes*.

CHAPTER 2

Running Autopush

If you just want to run autopush, for testing Push locally with Firefox, or to deploy autopush to a production environment for Firefox.

Architecture

Overview

For Autopush, we will focus on the section in the above diagram in the *Autopush* square.

Autopush consists of two types of server daemons:

`autopush` (connection node)

> Run a connection node. These handle large amounts of user agents (Firefox) using the Websocket protocol.

`autoendpoint` (endpoint node)

> Run an endpoint node. These provide a *WebPush* HTTP API for *Application Servers* to HTTP POST messages to endpoints.

To have a running Push Service for Firefox, both of these server daemons must be running and communicating with the same DynamoDB tables. A local DynamoDB can be run or AWS DynamoDB.

Endpoint nodes handle all *Notification* POST requests, looking up in DynamoDB to see what Push server the UAID is connected to. The Endpoint nodes then attempt delivery to the appropriate connection node. If the UAID is not online, the message may be stored in DynamoDB in the appropriate message table.

Push connection nodes accept websocket connections (this can easily be HTTP/2 for WebPush), and deliver notifications to connected clients. They check DynamoDB for missed notifications as necessary.

There will be many more Push servers to handle the connection node, while more Endpoint nodes can be handled as needed for notification throughput.

Cryptography

The HTTP endpoint URL's generated by the connection nodes contain encrypted information, the *UAID* and *Subscription* to send the message to. This means that they both must have the same `CRYPTO_KEY` supplied to each.

See *`make_endpoint()`* for the endpoint URL generator.

If you are only running Autopush locally, you can skip to *Running Autopush* as later topics in this document apply only to developing or production scale deployments of Autopush.

DynamoDB Tables

Autopush uses a single router table and multiple messages tables, one for each month of the year. On startup, Autopush will create the router table and a message table for the prior month and the current month of the year.

For more information on DynamoDB tables, see http://docs.aws.amazon.com/amazondynamodb/latest/gettingstartedguide/Welcome.html

Router Table Schema

The router table stores metadata for a given *UAID* as well as which month table should be used for clients with a `router_type` of `webpush`.

For *Bridging*, additional bridge-specific data may be stored in the router record for a *UAID*.

uaid	**partition key** - *UAID*
router_type	*Router Type*
node_id	Hostname of the connection node the client is connected to.
connected_at	Precise time (in milliseconds) the client connected to the node.
last_connect	**global secondary index** - year-month-hour that the client has last connected.
curmonth	Message table name to use for storing *WebPush* messages.

Autopush uses an optimistic deletion policy for `node_id` to avoid delete calls when not needed. During a delivery attempt, the endpoint will check the `node_id` for the corresponding *UAID*. If the client is not connected, it will clear the `node_id` record for that *UAID* in the router table.

If an endpoint node discovers during a delivery attempt that the `node_id` on record does not have the client connected, it will clear the `node_id` record for that *UAID* in the router table.

The `last_connect` has a secondary global index on it to allow for maintenance scripts to locate and purge stale client records and messages.

Clients with a `router_type` of `webpush` drain stored messages from the message table named `curmonth` after completing their initial handshake. If the `curmonth` entry is not the current month then it updates it to store new messages in the latest message table after stored message retrieval.

Message Table Schema

The message table stores messages for users while they're offline or unable to get immediate message delivery.

uaid	**partition key** - *UAID*
chidmessageid	**sort key** - *CHID* + *Message-ID*.
chids	Set of *CHID* that are valid for a given user. This entry is only present in the item when `chidmessageid` is a space.
data	Payload of the message, provided in the Notification body.
headers	HTTP headers for the Notification.
ttl	Time-To-Live for the Notification.
timestamp	Time (in seconds) that the message was saved.
updateid	UUID generated when the message is stored to track if the message is updated between a client reading it and attempting to delete it.

The subscribed channels are stored as `chids` in a record stored with a blank space set for `chidmessageid`. Before storing or delivering a *Notification* a lookup is done against these `chids`.

Message Table Rotation

To avoid costly table scans, autopush uses a rotating message and router table. Clients that haven't connected in 30-60 days will have their router and message table entries dropped and need to re-register.

Tables are post-fixed with the year/month they are meant for, ie:

> messages-2015-02

Tables must be created and have their read/write units properly allocated by a separate process in advance of the month switch-over as autopush nodes will assume the tables already exist. Scripts are provided that can be run weekly to ensure all necessary tables are present, and tables old enough are dropped.

See also:

Table maintenance script: https://github.com/mozilla-services/autopush/blob/master/maintenance.py

Within a few days of the new month, the load on the prior months table will fall as clients transition to the new table. The read/write units on the prior month may then be lowered.

Message Table Interaction Rules

Due to the complexity of having notifications spread across two tables, several rules are used to avoid losing messages during the month transition.

The logic for connection nodes is more complex, since only the connection node knows when the client connects, and how many messages it has read through.

The router table uses the `curmonth` field to indicate the last month the client has read notifications through. This is independent of the last_connect since it is possible for a client to connect, fail to read its notifications, then reconnect. This field is updated for a new month when the client connects **after** it has ack'd all the notifications out of the last month.

To avoid issues with time synchronization, the node the client is connected to acts as the source of truth for when the month has flipped over. Clients are only moved to the new table on connect, and only after reading/acking all the notifications for the prior month.

Rules for Endpoints

1. Check the router table to see the current_month the client is on.
2. Read the chan list entry from the appropriate month message table to see if its a valid channel.

   If its valid, move to step 3.

3. Store the notification in the current months table if valid. (Note that this step does not copy the blank entry of valid channels)

Rules for Connection Nodes

After Identification:

1. Check to see if the current_month matches the current month, if it does then proceed normally using the current months message table.

   If the connection node month does not match stored current_month in the clients router table entry, proceed to step 2.

2. Read notifications from prior month and send to client.

   Once all ACKs are received for all the notifications for that month proceed to step 3.

3. Copy the blank message entry of valid channels to the new month message table.

4. Update the router table for the current_month.

During switchover, only after the router table update are new commands from the client accepted.

Handling of Edge Cases:

- Connection node gets more notifications during step 3, enough to buffer, such that the endpoint starts storing them in the previous current_month. In this case the connection node will check the old table, then the new table to ensure it doesn't lose message during the switch.
- Connection node dies, or client disconnects during step 3/4. Not a problem as the reconnect will pick it up at the right spot.

Push Characteristics

- When the Push server has sent a client a notification, no further notifications will be accepted for delivery (except in one edge case). In this state, the Push server will reply to the Endpoint with a 503 to indicate it cannot currently deliver the notification. Once the Push server has received ACKs for all sent notifications, new notifications can flow again, and a check of storage will be done if the Push server had to reply with a 503. The Endpoint will put the Notification in storage in this case.
- (Edge Case) Multiple notifications can be sent at once, if a notification comes in during a Storage check, but before it has completed.
- If a connected client is able to accept a notification, then the Endpoint will deliver the message to the client completely bypassing Storage. This Notification will be referred to as a Direct Notification vs. a Stored Notification.
- Provisioned Write Throughput for the Router table determines how many connections per second can be accepted across the entire cluster.
- Provisioned Read Throughput for the Router table *and* Provisioned Write throughput for the Storage table determine maximum possible notifications per second that can be handled. In theory notification throughput can be higher than Provisioned Write Throughput on the Storage as connected clients will frequently not require using Storage at all. Read's to the Router table are still needed for every notification, whether Storage is hit or not.
- Provisioned Read Throughput on for the Storage table is an important factor in maximum notification throughput, as many slow clients may require frequent Storage checks.
- If a client is reconnecting, their Router record will be old. Router records have the node_id cleared optimistically by Endpoints when the Endpoint discovers it cannot deliver the notification to the Push node on file. If the conditional delete fails, it implies that the client has during this period managed to connect somewhere again. It's

entirely possible that the client has reconnected and checked storage before the Endpoint stored the Notification, as a result the Endpoint must read the Router table again, and attempt to tell the node_id for that client to check storage. Further action isn't required, since any more reconnects in this period will have seen the stored notification.

Push Endpoint Length

The Endpoint URL may seem excessively long. This may seem needless and confusing since the URL consists of the unique User Agent Identifier (UAID) and the Subscription Channel Identifier (CHID). Both of these are class 4 Universally Unique Identifiers (UUID) meaning that an endpoint contains 256 bits of entropy (2 * 128 bits). When used in string format, these UUIDs are always in lower case, dashed format (e.g. "01234567-0123-abcd-0123-0123456789ab").

Unfortunately, since the endpoint contains an identifier that can be easily traced back to a specific device, and therefore a specific user, there is the risk that a user might inadvertently disclose personal information via their metadata. To prevent this, the server obscures the UAID and CHID pair to prevent casual determination.

As an example, it is possible for a user to get a Push endpoint for two different accounts from the same User Agent. If the UAID were disclosed, then a site may be able to associate a single user to both of those accounts. In addition, there are reasons that storing the UAID and CHID in the URL makes operating the server more efficient.

Naturally, we're always looking at ways to improve and reduce the length of the URL. This is why it's important to store the entire length of the endpoint URL, rather than try and optimize in some manner.

Running Autopush

Overview

To run Autopush, you will need to run at least one connection node, one endpoint node, and a local DynamoDB server or AWS DynamoDB. The prior section on Autopush architecture documented these components and their relation to each other.

The recommended way to run the latest development or tagged Autopush release is to use docker. Autopush has docker images built automatically for every tagged release and when code is merged to master.

If you want to run the latest Autopush code from source then you should follow the *Developing Autopush* instructions.

The instructions below assume that you want to run Autopush with a local DynamoDB server for testing or local verification. The docker containers can be run on separate hosts as well, or with AWS DynamoDB instead.

Setup

These instructions will yield a locally running Autopush setup with the connection node listening on localhost port `8080`, with the endpoint node listening on localhost port `8082`. Make sure these ports are available on localhost before running, or change the configuration to have the Autopush daemons use other ports.

1. Install docker
2. Install docker-compose
3. Create a directory for your docker and Autopush configuration:

```
$ mkdir autopush-config
$ cd autopush-config
```

4. Fetch the latest `docker-compose.yml` file:

```
$ curl -O https://raw.githubusercontent.com/mozilla-services/autopush/
↪master/docker-compose.yml
```

5. Fetch the latest `boto-compose.cfg` file:

```
$ curl -O https://raw.githubusercontent.com/mozilla-services/autopush/
↪master/boto-compose.cfg
```

The `boto-compose.cfg` file will be mounted inside the Autopush docker containers when running to point Autopush at the locally running DynamoDB

Note: The docker images used take approximately 1.5 GB of disk-space, make sure you have appropriate free-space before proceeding.

Generate a Crypto-Key

As the *Cryptography* section notes, you will need a `CRYPTO_KEY` to run both of the Autopush daemons. To generate one with the docker image:

```
$ docker run -t -i ~/autopush autokey
Key = hkclU1V37Dnp-0DMF9HLe_40Nnr8kDTYVbo2yxuylzk=
```

Store the key for later use (including the trailing =).

Start Autopush

Once you've completed the setup and have a crypto key, you can run a local Autopush with a single command:

```
$ CRYPTO_KEY=hkclU1V37Dnp-0DMF9HLe_40Nnr8kDTYVbo2yxuylzk= docker-compose up
```

docker-compose will start up three containers, two for each Autopush daemon, and a third for DynamoDB.

By default, the following services will be exposed:

`ws://localhost:8080/` - websocket server

`http://localhost:8082/` - HTTP Endpoint Server (See *the HTTP API*)

You could set the `CRYPTO_KEY` as an environment variable, or setup a more thorough configuration using config files as documented below.

The load-tester can be run against it or you can run Firefox with the local Autopush per the *Firefox Testing* docs.

Configuration

Autopush can be configured in three ways; by option flags, by environment variables, and by configuration files. Autopush uses three configuration files. These files use standard *ini* formatting similar to the following:

```
# A comment description
;a_disabled_option
;another_disabled_option=default_value
option=value
```

Options can either have values or act as boolean flags. If the option is a flag it is either True if enabled, or False if disabled. The configuration files are usually richly commented, and you're encouraged to read them to learn how to set up your installation of autopush.

Please note: any line that does not begin with a # or *;* is considered an option line. if an unexpected option is present in a configuration file, the application will fail to start.

Configuration files can be located in:

- in the /etc/ directory
- in the configs subdirectory
- in the $HOME or current directory (prefixed by a period '.')

The three configuration files are:

- *autopush_connection.ini* - contains options for use by the websocket handler. This file's path can be specified by the `--config-connection` option.
- *autopush_shared.ini* - contains options shared between the connection and endpoint handler. This file's path can be specified by the `--config-shared` option.
- *autopush_endpoint.ini* - contains options for the HTTP handlers This file's path can be specified by the `--config-endpoint` option.

Sample Configurations

Three sample configurations, a base config, and a config for each Autopush daemon can be found at https://github.com/mozilla-services/autopush/tree/master/config

These can be downloaded and modified as desired.

Config Files with Docker

To use a configuration file with docker, ensure the config files are accessible to the user running docker-compose. Then you will need to update the `docker-compose.yml` to use the config files and make them available to the appropriate docker containers.

Mounting a config file to be available in a docker container is fairly simple, for instance, to mount a local file `autopush_connection.ini` into a container as `/etc/autopush_connection.ini`, update the `autopush` section of the `docker-compose.yml` to be:

```
volumes:
  - ./boto-compose.cfg:/etc/boto.cfg:ro
  - ./autopush_connection.ini:/etc/autopush_connection.ini
```

Autopush automatically searches for a configuration file at this location so nothing else is needed.

Notes on GCM/FCM support

Autopush is capable of routing messages over Google Cloud Messaging/Firebase Cloud Messaging for android devices. You will need to set up a valid GCM / FCM account. Once you have an account open the Google Developer Console:

- create a new project. Record the Project Number as "SENDER_ID". You will need this value for your android application.

- create a new Auth Credential Key for your project. This is available under **APIs & Auth >> Credentials** of the Google Developer Console. Store this value as `gcm_apikey` or `fcm_apikey` (as appropriate) in `.autopush_endpoint` server configuration file.
- add `gcm_enabled` to the `.autopush_shared` server configuration file to enable GCM routing.
- add `fcm_enabled` to the `.autopush_shared` server configuration file to enable FCM routing.

Additional notes on using the GCM/FCM bridge are available on the wiki.

CHAPTER 3

Developing Autopush

For developers wishing to work with the latest autopush source code, it's recommended that you first familiarize yourself with *running Autopush* before proceeding.

Installing

System Requirements

Autopush requires the following to be installed. Since each system has different methods and package names, it's best to search for each package.

- **Python 2.7.7 (or later 2.7.x), either**
  - PyPy 5.0.1 or later **or**
  - **CPython compiled with the following flags:**
    - –enable-unicode=usc4 –enable-ipv6
- **build-essential (a meta package that includes):**
  - autoconf
  - automake
  - gcc
  - make
- pypy **or** python (CPython) development (header files)
- libffi development
- openssl development
- python virtualenv
- git

For instance, if installing on a Fedora or RHEL-like Linux (e.g. an Amazon EC2 instance):

```
$ sudo yum install autoconf automake gcc make libffi-devel \
openssl-devel pypy pypy-devel python-virtualenv git -y
```

Or a Debian based system (like Ubuntu):

```
$ sudo apt-get install build-essential libffi-dev \
libssl-dev pypy-dev python-virtualenv git --assume-yes
```

Autopush uses the Boto python library. Be sure to properly set up your boto config file.

Notes on OS X

autopush depends on the Python cryptography library, which requires OpenSSL. If you're installing autopush on OS X with a custom version of OpenSSL, you'll need to set the `ARCHFLAGS` environment variable, and add your OpenSSL library path to `LDFLAGS` and `CFLAGS` before running `make`:

```
export ARCHFLAGS="-arch x86_64"
# Homebrew installs OpenSSL to `/usr/local/opt/openssl` instead of
# `/usr/local`.
export LDFLAGS="-L/usr/local/lib" CFLAGS="-I/usr/local/include"
```

Check-out the Autopush Repository

You should now be able to check-out the autopush repository.

```
$ git clone https://github.com/mozilla-services/autopush.git
```

Alternatively, if you're planning on submitting a patch/pull-request to autopush then fork the repo and follow the *Github Workflow* documented in Mozilla Push Service - Code Development.

Python 2.7.7+ w/virtualenv

You will need `virtualenv` installed per the above requirements. Set up your virtual environment by running the following (if using PyPy, you'll likely need to specify the `-p <path to pypy>` option):

```
$ virtualenv -p `which pypy` .
```

Then run the Makefile with `make` to setup the application.

Scripts

After installation of autopush the following command line utilities are available in the virtualenv `bin/` directory:

`autopush`	Runs a Connection Node
`autoendpoint`	Runs an Endpoint Node
`endpoint_diagnostic`	Runs Endpoint diagnostics
`autokey`	Endpoint encryption key generator

You will need to have a boto config file file or `AWS` environment keys setup before the first 3 utilities will run properly.

Building Documentation

To build the documentation, you will need additional packages installed:

```
$ pip install -r doc-requirements.txt
```

You can then build the documentation:

```
$ cd docs
$ make html
```

Using a Local DynamoDB Server

Amazon supplies a Local DynamoDB Java server to use for local testing that implements the complete DynamoDB API. This is used for automated unit testing on Travis and can be used to run autopush locally for testing.

You will need the Java JDK 6.x or newer.

To setup the server locally:

```
$ mkdir ddb
$ curl -sSL http://dynamodb-local.s3-website-us-west-2.amazonaws.com/dynamodb_local_
↪latest.tar.gz | tar xzvC ddb/
$ java -Djava.library.path=./ddb/DynamoDBLocal_lib -jar ./ddb/DynamoDBLocal.jar -
↪sharedDb -inMemory
```

An example boto config file is provided in `automock/boto.cfg` that directs autopush to your local DynamoDB instance.

Configuring for the APNS bridge

APNS requires a current Apple Developer License for the platform or platforms you wish to bridge to (e.g. iOS, desktop, etc.). Once that license has been acquired, you will need to create and export a valid *.p12* type key file. For this document, we will concentrate on creating an iOS certificate.

Create the App ID

First, you will need an Application ID. If you do not already have an application, you will need to create an application ID. For an App ID to use Push Notifications, it must be created as an **Explicit App ID**. Please be sure that under "**App Services**" you select **Push Notifications**. Once these values are set, click on [Continue].

Confirm that the app settings are as you desire and click [Register], or click [Back] and correct them. **Push Notifications** should appear as "Configurable".

Create the Certificate

Then Create a new certificate. Select "Apple Push Notification service SSL" for either Development or Production, depending on intended usage of the certificate. "Development", in this case, means a certificate that will not be used by an application released for general public use, but instead only for personal or team development. This is also known as a "Sandbox" application and will require setting the "use_sandbox" flag. Once the preferred option is selected, click [Continue].

Select the App ID that matches the Application that will use Push Notifications. Several Application IDs may be present, be sure to match the correct App ID. This will be the App ID which will act as the recipient bridge for Push Notifications. Select [Continue].

Follow the on-screen instructions to generate a **CSR file**, click [Continue], and upload the CSR.

Download the newly created *iOSTeam_Provisioning_Profile_.mobileprovision* keyset, and import it into your **KeyChain Access** app.

Exporting the .p12 key set

In **KeyChain Access**, for the **login** keychain, in the **Certificates** category, you should find an **Apple Push Services: *your AppID*** certificate. Right click on this certificate and select *Export "Apple Push Services:"...*. Provide the file with a reasonably unique name, such as "Push_Production_APNS_Keys.p12", so that you can find it easily later. You may wish to secure these keys with a password.

Converting .p12 to PEM

You will need to convert the .p12 file to PEM format. *openssl* can perform these steps for you. A simple script you could use might be:

```
#!/bin/bash
echo Converting $1 to PEM
openssl pkcs12 -in $1 -out $1_cert.pem -clcerts -nokeys
openssl pkcs12 -in $1 -out $1_key.pem -nocerts -nodes
```

This will divide the p12 key into two components that can be read by the autopush application.

Sending the APNS message

The APNS post message contains JSON formatted data similar to the following:

```
{
    "aps": {
        "content-available": 1
    },
    "key": "value",
    ...
}
```

aps is reserved as a sub-dictionary. All other *key*: *value* slots are open.

In addition, you must specify the following headers:

- *apns-id*: A lowercase, dash formatted UUID for this message.
- *apns-priority*: Either **10** for Immediate delivery or **5** for delayable delivery.
- *apns-topic*: The bundle ID for the recipient application. This must match the bundle ID of the AppID used to create the *"Apple Push Services:..."* certificate. It usually has the format of *com.example.ApplicationName*.
- *apns-expiration*: The timestamp for when this message should expire in UTC based seconds. A zero ("0") means immediate expiration.

Handling APNS responses

APNS returns a status code and an optional JSON block describing the error. A list of these responses are provided in the APNS documentation (Note, Apple may change the document locaiton without warning. you may be able to search using DeviceTokenNotForTopic or similar error messages.)

Testing

Running Tests

If you plan on doing development and testing, you will need to install some additional packages.

```
$ bin/pip install -r test-requirements.txt
```

Once the Makefile has been run, you can run `make test` to run the test suite.

Note: Failures may occur if a `.boto` file exists in your home directory. This file should be moved elsewhere before running the tests.

Disabling Integration Tests

`make test` runs the `tox` program which can be difficult to break for debugging purposes. The following bash script has been useful for running tests outside of tox:

```
#! /bin/bash
mv autopush/tests/test_integration.py{,.hold}
mv autopush/tests/test_logging.py{,.hold}
bin/nosetests -sv autopush
mv autopush/tests/test_integration.py{.hold,}
mv autopush/tests/test_logging.py{.hold,}
```

This script will cause the integration and logging tests to not run.

Firefox Testing

To test a locally running Autopush with Firefox, you will need to edit several config variables in Firefox.

1. Open a New Tab.
2. Go to `about:config` in the Location bar and hit Enter, accept the disclaimer if it's shown.
3. Search for `dom.push.serverURL`, make a note of the existing value (you can right-click the preference and choose `Reset` to restore the default).
4. Double click the entry and change it to `ws://localhost:8080/`.
5. Right click in the page and choose `New -> Boolean`, name it `dom.push.testing.allowInsecureServerURL` and set it to `true`.

You should then restart Firefox to begin using your local Autopush.

Debugging

On Android, you can set `dom.push.debug` to enable debug logging of Push via `adb logcat`.

For desktop use, you can set `dom.push.loglevel` to `"debug"`. This will log all push messages to the Browser Console (Tools > Web Developer > Browser Console).

Release Process

Autopush has a regular 2-3 week release to production depending on developer and QA availability. The developer creating a release should handle all aspects of the following process as they're done closely in order and time.

Versions

Autopush uses a `{major}.{minor}.{patch}` version scheme, new `{major}` versions are only issued if backwards compatibility is affected. Patch versions are used if a critical bug occurs after production deployment that requires a bug fix immediately.

Dev Releases

When changes are committed to the `master` branch, an operations Jenkins instance will build and deploy the code automatically to the dev environment.

The development environment can be verified at its endpoint/wss endpoints:

- Websocket: wss://autopush.dev.mozaws.net/
- Endpoint: https://updates-autopush.dev.mozaws.net/

Stage/Production Releases

Pre-Requisites

To create a release, you will need appropriate access to the autopush GitHub repository with push permission.

You will also need clog installed to create the `CHANGELOG.md` update.

Release Steps

In these steps, the `{version}` refers to the full version of the release.

i.e. If a new minor version is being released after `1.21.0`, the `{version}` would be `1.22.0`.

1. Switch to the `master` branch of autopush.
2. `git pull` to ensure the local copy is completely up-to-date.
3. `git diff origin/master` to ensure there are no local staged or uncommited changes.
4. Run `tox` locally to ensure no artifacts or other local changes that might break tests have been introduced.

5. Change to the release branch.

   If this is a new major/minor release, `git checkout -b release/{major}.{minor}` to create a new release branch.

   If this is a new patch release, you will first need to ensure you have the minor release branch checked out, then:

   (a) `git checkout release/{major}.{minor}`

   (b) `git pull` to ensure the branch is up-to-date.

   (c) `git merge master` to merge the new changes into the release branch.

   Note that the release branch does not include a ‘‘{patch}‘‘ component.

6. Edit `autopush/__init__.py` so that the version number reflects the desired release version.
7. Run `clog --setversion {version}`, verify changes were properly accounted for in `CHANGELOG.md`.
8. `git add CHANGELOG.md autopush/__init__.py` to add the two changes to the new release commit.
9. `git commit -m "chore: tag {version}"` to commit the new version and record of changes.
10. `git tag -s -m "chore: tag {version}" {version}` to create a signed tag of the current HEAD commit for release.
11. `git push --set-upstream origin release/{major}.{minor}` to push the commits to a new origin release branch.
12. `git push --tags origin release/{major}.{minor}` to push the tags to the release branch.
13. Submit a pull request on github to merge the release branch to master.
14. Go to the autopush releases page, you should see the new tag with no release information under it.
15. Click the `Draft a new release` button.
16. Enter the tag for `Tag version`.
17. Copy/paste the changes from `CHANGELOG.md` into the release description omitting the top 2 lines (the a name HTML and the version) of the file.

    Keep these changes handy, you’ll need them again shortly.

18. Once the release branch pull request is approved and merged, click `Publish Release`.
19. File a bug for stage deployment in Bugzilla, in the `Cloud Services` product, under the `Operations: Deployment Requests` component. It should be titled `Please deploy autopush {major}.{minor} to STAGE` and include the changes in the Description along with any additional instructions to operations regarding deployment changes and special test cases if needed for QA to verify.

At this point, QA will take-over, verify stage, and create a production deployment Bugzilla ticket. QA will also schedule production deployment for the release.

Coding Style Guide

Autopush uses Python styling guides based on PEP8 and PEP257.

Exceptions

- Single sentence docstrings are formatted the same way as a single line docstring, but may not always include ending punctuation.

- File level docstrings may not include a line break before the first line of code.

CHAPTER 4

Source Code

All source code is available on github under autopush.

Code Documentation

Code Documentation

Comprehensive code documentation for *autopush* is available within. The code documentation is organized alphabetically by module name.

autopush.db

Database Interaction

WebPush Sort Keys

Messages for WebPush are stored using a partition key + sort key, originally the sort key was:

    CHID : Encrypted(UAID: CHID)

The encrypted portion was returned as the Location to the Application Server. Decrypting it resulted in enough information to create the sort key so that the message could be deleted and located again.

For WebPush Topic messages, a new scheme was needed since the only way to locate the prior message is the UAID + CHID + Topic. Using Encryption in the sort key is therefore not useful since it would change every update.

The sort key scheme for WebPush messages is:

    VERSION : CHID : TOPIC

To ensure updated messages are not deleted, each message will still have an update-id key/value in its item.

Non-versioned messages are assumed to be original messages from before this scheme was adopted.

`VERSION` is a 2-digit 0-padded number, starting at 01 for Topic messages.

DynamoDB Table Functions

autopush.db.**create_router_table**(*tablename='router'*, *read_throughput=5*, *write_throughput=5*)
Create a new router table

The last_connect index is a value used to determine the last month a user was seen in. To prevent hot-keys on this table during month switchovers the key is determined based on the following scheme:

(YEAR)(MONTH)(DAY)(HOUR)(0001-0010)

Note that the random key is only between 1-10 at the moment, if the key is still too hot during production the random range can be increased at the cost of additional queries during GC to locate expired users.

autopush.db.**create_storage_table**(*tablename='storage'*, *read_throughput=5*, *write_throughput=5*)
Create a new storage table for simplepush style notification storage

autopush.db.**get_router_table**(*tablename='router'*, *read_throughput=5*, *write_throughput=5*)
Get the main router table object

Creates the table if it doesn't already exist, otherwise returns the existing table.

autopush.db.**get_storage_table**(*tablename='storage'*, *read_throughput=5*, *write_throughput=5*)
Get the main storage table object

Creates the table if it doesn't already exist, otherwise returns the existing table.

Utility Functions

autopush.db.**preflight_check**(*storage*, *router*, *uaid='deadbeef00000000deadbeef00000000'*)
Performs a pre-flight check of the storage/router/message to ensure appropriate permissions for operation.

Failure to run correctly will raise an exception.

DynamoDB Table Class Abstractions

class autopush.db.**Storage**(*table*, *metrics*)
Create a Storage table abstraction on top of a DynamoDB Table object

__init__(*table*, *metrics*)
Create a new Storage object

Parameters

- **table** – Table object.
- **metrics** – Metrics object that implements the *autopush.metrics.IMetrics* interface.

fetch_notifications(**args*, ***kwargs*)
Fetch all notifications for a UAID

Raises ProvisionedThroughputExceededException if dynamodb table exceeds throughput.

save_notification(**args*, ***kwargs*)
Save a notification for the UAID

Raises ProvisionedThroughputExceededException if dynamodb table exceeds throughput.

delete_notification(*uaid*, *chid*, *version=None*)
Delete a notification for a UAID

Returns Whether or not the notification was able to be deleted.

class autopush.db.**Router**(*table*, *metrics*)
Create a Router table abstraction on top of a DynamoDB Table object

__init__(*table*, *metrics*)
Create a new Router object

Parameters

- **table** – Table object.
- **metrics** – Metrics object that implements the *autopush.metrics.IMetrics* interface.

get_uaid(*uaid*)
Get the database record for the UAID

Raises ItemNotFound if there is no record for this UAID. ProvisionedThroughputExceededException if dynamodb table exceeds throughput.

register_user(**args*, ***kwargs*)
Register this user

If a record exists with a newer connected_at, then the user will not be registered.

Returns Whether the user was registered or not.

Raises ProvisionedThroughputExceededException if dynamodb table exceeds throughput.

drop_user(**args*, ***kwargs*)
Drops a user record

delete_uaids(*uaids*)
Issue a batch delete call for the given uaids

drop_old_users(*months_ago=2*)
Drops user records that have no recent connection

Utilizes the last_connect index to locate users that haven't connected in the given time-frame.

The caller must iterate through this generator to trigger batch delete calls. Caller should wait as appropriate to avoid exceeding table limits.

Each iteration will result in a batch delete for the currently iterated batch. This implies a set of writes equal in size to the 25 * record-size minimum.

Warning: Calling list() on this generator will likely exceed provisioned write through-put as the batch-delete calls will be made as quickly as possible.

Parameters **months_ago** – how many months ago since the last connect

Returns Iterable of how many deletes were run

update_message_month(**args*, ***kwargs*)
Update the route tables current_message_month

Note that we also update the last_connect at this point since webpush users when connecting will always call this once that month. The current_timestamp is also reset as a new month has no last read timestamp.

clear_node(**args*, ***kwargs*)
Given a router item and remove the node_id

The node_id will only be cleared if the `connected_at` matches up with the item's `connected_at`.

Returns Whether the node was cleared or not.

Raises `ProvisionedThroughputExceededException` if dynamodb table exceeds throughput.

autopush.exceptions

Autopush Exceptions

class `autopush.exceptions.`**AutopushException**
Parent Autopush Exception

class `autopush.exceptions.`**RouterException**(*message*, *status_code=500*, *response_body=''*, *router_data=None*, *headers=None*, *log_exception=True*, *errno=None*, *logged_status=None*, ***kwargs*)
Exception if routing has failed, may include a custom status_code and body to write to the response.

__init__(*message*, *status_code=500*, *response_body=''*, *router_data=None*, *headers=None*, *log_exception=True*, *errno=None*, *logged_status=None*, ***kwargs*)
Create a new RouterException

autopush.logging

Custom Logging Setup

class `autopush.logging.`**PushLogger**(*logger_name*, *log_level='debug'*, *log_format='json'*, *log_output='stdout'*, *sentry_dsn=None*, *firehose_delivery_stream=None*)
Twisted LogObserver implementation

Supports firehose delivery, Raven exception reporting, and json/test console debugging output.

class `autopush.logging.`**FirehoseProcessor**(*stream_name*, *maxsize=0*)
Batches log events for sending to AWS FireHose

autopush.main

autopush/autoendpoint daemon scripts

Daemon Script Entry Points

class `autopush.main.`**ConnectionApplication**(**args*, ***kwargs*)
The autopush application

static parse_args(*config_files*, *args*)
Parse out connection node arguments for an autopush node

websocket_factory
alias of PushServerFactory

websocket_site_factory
alias of ConnectionWSSite

add_internal_router()
Start the internal HTTP notification router

add_websocket()
Start the public WebSocket server

class autopush.main.**EndpointApplication**(**args*, ***kwargs*)
The autoendpoint application

static parse_args(*config_files*, *args*)
Parses out endpoint arguments for an autoendpoint node

add_endpoint()
Start the Endpoint HTTP router

Common Root

class autopush.main.**AutopushMultiService**(*settings*)

static parse_args(*config_files*, *args*)
Parse command line args via argparse

setup(*rotate_tables=True*)
Initialize the services

add_maybe_ssl(*port*, *factory*, *ssl_cf*)
Add a Service from factory, optionally behind TLS

add_timer(**args*, ***kwargs*)
Add a TimerService

add_memusage()
Add the memusage Service

run()
Start the services and run the reactor

classmethod _from_argparse(*ns*, ***kwargs*)
Create an instance from argparse/additional kwargs

classmethod main(*args=None*, *use_files=True*)
Entry point to autopush's main command line scripts.

aka autopush/autoendpoint.

Utility Functions

autopush.metrics

Metrics interface and implementations

Interface

class autopush.metrics.**IMetrics**(**args*, ***kwargs*)
Metrics interface

Each method except *__init__()* and *start()* must be implemented.

Additional `kwargs` may be recorded as additional metric tags for metric systems that support it, otherwise they should be ignored.

__init__(**args*, ***kwargs*)
Setup the metrics

start()
Start any connection needed for metric transmission

increment(*name*, *count=1*, ***kwargs*)
Increment a counter for a metric name

gauge(*name*, *count*, ***kwargs*)
Record a gauge for a metric name

timing(*name*, *duration*, ***kwargs*)
Record a timing in ms for a metric name

Implementations

class autopush.metrics.**SinkMetrics**(**args*, ***kwargs*)
Exists to ignore metrics when metrics are not active

class autopush.metrics.**TwistedMetrics**(*statsd_host='localhost'*, *statsd_port=8125*)
Twisted implementation of statsd output

class autopush.metrics.**DatadogMetrics**(*api_key*, *app_key*, *hostname*, *flush_interval=10*, *namespace='autopush'*)
DataDog Metric backend

autopush.protocol

Basic Protocol for ignoring data

class autopush.protocol.**IgnoreBody**(*response*, *deferred*)
A protocol that discards any data it receives

This is necessary to support persistent HTTP connections. If the response body is never read using `Response.deliverBody`, or `stopProducing()` is called, the connection will not be reused.

classmethod ignore(*response*)
Class method helper for ignoring the response

dataReceived(*data*)
Ignore received data

connectionLost(*reason*)
Relay back the loss of connection to the deferred

autopush.router.apnsrouter

APNS Router

class autopush.router.apnsrouter.**APNSRouter**(*ap_settings*, *router_conf*, *metrics*, *load_connections=True*)

APNS Router Implementation

_connect(*rel_channel*, *load_connections=True*)

Connect to APNS

Parameters

- **rel_channel** (*str*) – Release channel name (e.g. Firefox. FirefoxBeta,..)
- **load_connections** (*bool*) – (used for testing)

Returns APNs to be stored under the proper release channel name.

Return type apns.APNs

__init__(*ap_settings*, *router_conf*, *metrics*, *load_connections=True*)

Create a new APNS router and connect to APNS

Parameters

- **ap_settings** (autopush.settings.AutopushSettings) – Configuration settings
- **router_conf** (*dict*) – Router specific configuration
- **load_connections** (*bool*) – (used for testing)

register(*uaid*, *router_data*, *app_id*, **args*, ***kwargs*)

Register an endpoint for APNS, on the *app_id* release channel.

This will validate that an APNs instance token is in the *router_data*,

Parameters

- **uaid** – User Agent Identifier
- **router_data** – Dict containing router specific configuration info
- **app_id** – The release channel identifier for cert info lookup

amend_endpoint_response(*response*, *router_data*)

Stubbed out for this router

route_notification(*notification*, *uaid_data*)

Start the APNS notification routing, returns a deferred

Parameters

- **notification** (*autopush.endpoint.Notification*) – Notification data to send
- **uaid_data** (*dict*) – User Agent specific data

_route(*notification*, *router_data*)

Blocking APNS call to route the notification

Parameters

- **notification** (*dict*) – Notification data to send
- **router_data** (*dict*) – Pre-initialized data for this connection

class autopush.router.apns2.**APNSClient**(*cert_file*, *key_file*, *topic*, *alt=False*, *use_sandbox=False*, *max_connections=20*, *logger=None*, *metrics=None*, *load_connections=True*)

__init__(*cert_file*, *key_file*, *topic*, *alt=False*, *use_sandbox=False*, *max_connections=20*, *logger=None*, *metrics=None*, *load_connections=True*)

Create the APNS client connector.

The cert_file and key_file can be derived from the exported *.p12* **Apple Push Services: *bundleID* ** key contained in the **Keychain Access** application. To extract the proper PEM formatted data, you can use the following commands:

```
` openssl pkcs12 -in file.p12 -out apns_cert.pem -clcerts -nokeys
openssl pkcs12 -in file.p12 -out apns_key.pem -nocerts -nodes `
```

The *topic* is the Bundle ID of the bridge recipient iOS application. Since the cert needs to be tied directly to an application, the topic is usually similar to "com.example.MyApplication".

Parameters

- **cert_file** (*str*) – Path to the PEM formatted APNs certification file.
- **key_file** (*str*) – Path to the PEM formatted APNs key file.
- **topic** (*str*) – The *Bundle ID* that identifies the assoc. iOS app.
- **alt** (*bool*) – Use the alternate APNs publication port (if 443 is blocked)
- **use_sandbox** (*bool*) – Use the development sandbox
- **max_connections** (*int*) – Max number of pooled connections to use
- **logger** (*logger*) – Status logger
- **metrics** (*autopush.metrics.IMetric*) – Metric recorder
- **load_connections** (*bool*) – used for testing

send(*router_token*, *payload*, *apns_id*, *priority=True*, *topic=None*, *exp=None*)

Send the dict of values to the remote bridge

This sends the raw data to the remote bridge application using the APNS2 HTTP2 API.

Parameters

- **router_token** (*str*) – APNs provided hex token identifying recipient
- **payload** (*dict*) – Data to send to recipient
- **priority** (*bool*) – True is high priority, false is low priority
- **topic** (*str*) – BundleID for the recipient application (overides default)
- **exp** (*timestamp*) – Message expiration timestamp

autopush.router.gcm

GCM Router

class autopush.router.gcm.**GCMRouter**(*ap_settings*, *router_conf*, *metrics*)

GCM Router Implementation

__init__(*ap_settings*, *router_conf*, *metrics*)

Create a new GCM router and connect to GCM

register(*uaid*, *router_data*, *app_id*, **args*, ***kwargs*)
Validate that the GCM Instance Token is in the `router_data`

route_notification(*notification*, *uaid_data*)
Start the GCM notification routing, returns a deferred

_route(*notification*, *uaid_data*)
Blocking GCM call to route the notification

_error(*err*, *status*, ***kwargs*)
Error handler that raises the RouterException

_process_reply(*reply*, *uaid_data*, *ttl*, *notification*)
Process GCM send reply

autopush.router.fcm

FCM Router

class `autopush.router.fcm.`**FCMRouter**(*ap_settings*, *router_conf*, *metrics*)
FCM Router Implementation

Note: FCM is a newer branch of GCM. While there's not much change required for the server, there is significant work required for the client. To that end, having a separate router allows the "older" GCM to persist and lets the client determine when they want to use the newer FCM route.

__init__(*ap_settings*, *router_conf*, *metrics*)
Create a new FCM router and connect to FCM

register(*uaid*, *router_data*, *app_id*, **args*, ***kwargs*)
Validate that the FCM Instance Token is in the `router_data`

route_notification(*notification*, *uaid_data*)
Start the FCM notification routing, returns a deferred

_route(*notification*, *router_data*)
Blocking FCM call to route the notification

_error(*err*, *status*, ***kwargs*)
Error handler that raises the RouterException

_process_reply(*reply*, *notification*, *router_data*, *ttl*)
Process FCM send reply

autopush.router.interface

Router interface

class `autopush.router.interface.`**RouterResponse**(*status_code=200*, *response_body=''*, *router_data=None*, *headers=None*, *errno=None*, *logged_status=None*)
Router response if routing has succeeded.

If the router data needs to change as a result of this message, either the router got invalidated, or needs updating, then the router_data should be set.

__init__(*status_code=200*, *response_body=''*, *router_data=None*, *headers=None*, *errno=None*, *logged_status=None*)
Create a new RouterResponse

class autopush.router.interface.**IRouter**(*settings*, *router_conf*, ***kwargs*)

__init__(*settings*, *router_conf*, ***kwargs*)
Initialize the Router to handle notifications and registrations with the given settings and router conf.

register(*uaid*, *router_data*, *app_id*, **args*, ***kwargs*)
Register the uaid with router_data however is preferred prior to storing router_data for this user.

Parameters

- **uaid** – User Agent Identifier
- **router_data** – Route specific configuration info
- **app_id** – Application identifier from URI

Raises RouterException if data supplied is invalid.

amend_endpoint_response(*response*, *router_data*)
Modify an outbound Endpoint registration response to include router info.

Some routers require additional info to be returned to clients.

Parameters

- **response** – The response data to be sent to the client
- **router_data** – Route specific configuration info

route_notification(*notification*, *uaid_data*)
Route a notification

Parameters

- **notification** – A Notificaiton instance.
- **uaid_data** – A dict of the full user item from the db record.

Returns A response object upon successful routing.

Return type *RouterResponse*

Raises RouterException if routing fails.

This function runs in the main reactor, if a yield is needed then a deferred must be returned for the callback chain.

autopush.router.simple

Simple(Push) Style Autopush Router

This router handles notifications that should be dispatched to an Autopush node, or stores it appropriately in DynamoDB for SimplePush style version based channel ID's (only newest version is stored, no data stored).

class autopush.router.simple.**SimpleRouter**(*ap_settings*, *router_conf*, *db*, *agent*)
Implements *autopush.router.interface.IRouter* for internal routing to an Autopush node

__init__(*ap_settings*, *router_conf*, *db*, *agent*)
Create a new SimpleRouter

register(*uaid*, *router_data*, *app_id*, **args*, ***kwargs*)
No additional routing data

amend_endpoint_response(*response*, *router_data*)
Stubbed out for this router

route_notification(**args*, ***kwargs*)
Route a notification to an internal node, and store it if the node can't deliver immediately or is no longer a valid node

_save_notification(*uaid_data*, *notification*)
Saves a notification, returns a deferred.

This function is split out for the Webpush-style individual message storage to subclass and override.

_send_notification(*uaid*, *node_id*, *notification*)
Send a notification to a specific node_id

_send_notification_check(*uaid*, *node_id*)
Send a command to the node to check for notifications

_eat_db_err(*fail*)
errBack for ignoring provisioned throughput errors

autopush.web.base

class autopush.web.base.**ThreadedValidate**(*schema*)
A cyclone request validation decorator

Exposed as a classmethod for running a marshmallow-based validation schema in a separate thread for a cyclone request handler.

_validate_request(*request_handler*, **args*, ***kwargs*)
Validates a schema_class against a cyclone request

_track_validation_timing(*result*, *request_handler*, *start_time*)
Track the validation timing

classmethod validate(*schema*)
Validate a request schema in a separate thread before calling the request handler

An alias *threaded_validate* should be used from this module.

Using *cyclone.web.asynchronous* is not needed as this function will attach equivilant functionality to the method handler. Calling *self.finish()* is needed on decorated handlers.

Validated requests are deserialized into the **kwargs of the wrapped request handler method.

```
class MySchema(Schema):
    uaid = fields.UUID(allow_none=True)

class MyHandler(cyclone.web.RequestHandler):
    @threaded_validate(MySchema())
    def post(self, uaid=None):
        ...
```

class autopush.web.base.**Notification**(*version*, *data*, *channel_id*)
Parsed notification from the request

class autopush.web.base.**BaseWebHandler**(*application*, *request*, ***kwargs*)
Common overrides for Push web API's

initialize()
Setup basic aliases and attributes

prepare()
Common request preparation

options(**args*, ***kwargs*)
HTTP OPTIONS Handler

head(**args*, ***kwargs*)
HTTP HEAD Handler

_write_response(*status_code*, *errno*, *message=None*, *error=None*, *headers=None*, *url='http://autopush.readthedocs.io/en/latest/http.html#error-codes'*, *router_type=None*, *vapid=None*)
Writes out a full JSON error and sets the appropriate status

_validation_err(*fail*)
errBack for validation errors

_response_err(*fail*)
errBack for all exceptions that should be logged

This traps all exceptions to prevent any further callbacks from running.

_overload_err(*fail*)
errBack for throughput provisioned exceptions

_boto_err(*fail*)
errBack for random boto exceptions

_router_fail_err(*fail*, *router_type=None*, *vapid=False*)
errBack for router failures

_write_validation_err(*errors*)
Writes a set of validation errors out with details about what went wrong

_db_error_handling(*d*)
Tack on the common error handling for a dynamodb request and uncaught exceptions

_track_timing(*status_code=None*)
Logs out the request timing tracking stats

Note: The status code should be set before calling this function or passed in.

class autopush.web.base.**BaseWebHandler**(*application*, *request*, ***kwargs*)
Common overrides for Push web API's

initialize()
Setup basic aliases and attributes

prepare()
Common request preparation

options(**args*, ***kwargs*)
HTTP OPTIONS Handler

head(**args*, ***kwargs*)
HTTP HEAD Handler

_write_response(*status_code*, *errno*, *message=None*, *error=None*, *headers=None*, *url='http://autopush.readthedocs.io/en/latest/http.html#error-codes'*, *router_type=None*, *vapid=None*)
Writes out a full JSON error and sets the appropriate status

_validation_err(*fail*)
errBack for validation errors

_response_err(*fail*)
errBack for all exceptions that should be logged

This traps all exceptions to prevent any further callbacks from running.

_overload_err(*fail*)
errBack for throughput provisioned exceptions

_boto_err(*fail*)
errBack for random boto exceptions

_router_fail_err(*fail*, *router_type=None*, *vapid=False*)
errBack for router failures

_write_validation_err(*errors*)
Writes a set of validation errors out with details about what went wrong

_db_error_handling(*d*)
Tack on the common error handling for a dynamodb request and uncaught exceptions

_track_timing(*status_code=None*)
Logs out the request timing tracking stats

Note: The status code should be set before calling this function or passed in.

autopush.web.webpush

class autopush.web.webpush.**WebPushHandler**(*application*, *request*, ***kwargs*)

initialize()
Must run on initialization to set ahead of validation

_router_completed(*response*, *uaid_data*, *warning=''*, *router_type=None*, *vapid=None*)
Called after router has completed successfully

autopush.web.simplepush

class autopush.web.simplepush.**SimplePushHandler**(*application*, *request*, ***kwargs*)

initialize()
Must run on initialization to set ahead of validation

_router_completed(*response*, *uaid_data*, *warning=''*)
Called after router has completed successfully

autopush.web.log_check

class autopush.web.log_check.**LogCheckHandler**(*application*, *request*, ***kwargs*)

authenticate_peer_cert()
LogCheck skips authentication checks

get(*request_handler*, **args*, ***kwargs*)
HTTP GET

Generate a dummy error message for logging

autopush.web.message

class autopush.web.message.**MessageHandler**(*application*, *request*, ***kwargs*)

delete(*request_handler*, **args*, ***kwargs*)
Drops a pending message.

The message will only be removed from DynamoDB. Messages that were successfully routed to a client as direct updates, but not delivered yet, will not be dropped.

autopush.web.registration

class autopush.web.registration.**NewRegistrationHandler**(*application*, *request*, ***kwargs*)
Handle new bridge uaid registrations

post(*request_handler*, **args*, ***kwargs*)
HTTP POST

Router type/data registration.

_register_user_and_channel(*uaid*, *chid*, *router_type*, *router_data*)
Register a new user/channel, return its endpoint

class autopush.web.registration.**UaidRegistrationHandler**(*application*, *request*, ***kwargs*)
Handles UAID bridge methods

get(*request_handler*, **args*, ***kwargs*)
HTTP GET

Return a list of known channelIDs for a given UAID

put(*request_handler*, **args*, ***kwargs*)
HTTP PUT

Update router type/data for a UAID.

post(*request_handler*, **args*, ***kwargs*)
HTTP PUT

Update router type/data for a UAID.

delete(*request_handler*, **args*, ***kwargs*)
HTTP DELETE

Delete all pending records for the given UAID

_uaid_not_found_err(*fail*)
errBack for uaid lookup not finding the user

class autopush.web.registration.**SubRegistrationHandler**(*application*, *request*, ***kwargs*)
Handle a new channel for a bridge user

class autopush.web.registration.**ChannelRegistrationHandler**(*application*, *request*, ***kwargs*)
Handle deleting a channel for a bridge user

_chid_not_found_err(*fail*)
errBack for unknown chid

autopush.web.healthhandler

Health Check HTTP Handler

class autopush.web.health.**HealthHandler**(*application*, *request*, ***kwargs*)
HTTP Health Handler

authenticate_peer_cert()
Skip authentication checks

get(**args*, ***kwargs*)
HTTP Get

Returns basic information about the version and how many clients are connected in a JSON object.

_check_table(*table*)
Checks the tables known about in DynamoDB

_check_success(*result*, *name*)
Verifies a name is in the list of tables

_check_error(*failure*, *name*)
Returns an error, and why

_finish_response(*results*)
Returns whether the check succeeded or not

autopush.web.statushandler .. autoclass:: StatusHandler

members

private-members

member-order bysource

autopush.settings

Autopush Settings Object and Setup

class autopush.settings.**AutopushSettings**(*debug=False*, *crypto_key=None*, *bear_hash_key=NOTHING*, *hostname=None*, *port=None*, *resolve_hostname=False*, *router_scheme=None*, *router_hostname=None*, *router_port=None*, *endpoint_scheme=None*, *endpoint_hostname=None*, *endpoint_port=None*, *proxy_protocol_port=None*, *memusage_port=None*, *statsd_host='localhost'*, *statsd_port=8125*, *datadog_api_key=None*, *datadog_app_key=None*, *datadog_flush_interval=None*, *router_tablename='router'*, *router_read_throughput=5*, *router_write_throughput=5*, *storage_tablename='storage'*, *storage_read_throughput=5*, *storage_write_throughput=5*, *message_tablename='message'*, *message_read_throughput=5*, *message_write_throughput=5*, *preflight_uaid='deadbeef00000000deadbeef00000000'*, *ssl_key=None*, *ssl_cert=None*, *ssl_dh_param=None*, *router_ssl_key=None*, *router_ssl_cert=None*, *client_certs=None*, *router_conf=NOTHING*, *connect_timeout=0.5*, *max_data=4096*, *env='development'*, *ami_id=None*, *cors=False*, *hello_timeout=0*, *wake_timeout=0*, *msg_limit=100*, *auto_ping_interval=None*, *auto_ping_timeout=None*, *max_connections=None*, *close_handshake_timeout=None*, *notification_legacy=False*)

Main Autopush Settings Object

enable_tls_auth
Whether TLS authentication w/ client certs is enabled

classmethod from_argparse(*ns*, ***kwargs*)
Create an instance from argparse/additional kwargs

make_simplepush_endpoint(*uaid*, *chid*)
Create a simplepush endpoint

make_endpoint(*uaid*, *chid*, *key=None*)
Create an v1 or v2 WebPush endpoint from the identifiers.

Both endpoints use bytes instead of hex to reduce ID length. v1 is the uaid + chid v2 is the uaid + chid + sha256(key).bytes

Parameters

- **uaid** – User Agent Identifier
- **chid** – Channel or Subscription ID
- **key** – Optional Base64 URL-encoded application server key

Returns Push endpoint

parse_endpoint(*metrics*, *token*, *version='v1'*, *ckey_header=None*, *auth_header=None*)
Parse an endpoint into component elements of UAID, CHID and optional key hash if v2

Parameters

- **token** – The obscured subscription data.
- **version** – This is the API version of the token.
- **ckey_header** – the Crypto-Key header bearing the public key (from Crypto-Key: p256ecdsa=)
- **auth_header** – The Authorization header bearing the VAPID info

Raises **ValueError** – In the case of a malformed endpoint.

Returns a dict containing (uaid=UAID, chid=CHID, public_key=KEY)

autopush.ssl

Custom SSL configuration

class autopush.ssl.**AutopushSSLContextFactory**(**args*, ***kwargs*)
A SSL context factory

cacheContext()
Setup the main context factory with custom SSL settings

autopush.utils

A small collection of Autopush utility functions

autopush.utils.**canonical_url**(*scheme*, *hostname*, *port=None*)
Return a canonical URL given a scheme/hostname and optional port

autopush.utils.**resolve_ip**(*hostname*)
Resolve a hostname to its IP if possible

autopush.utils.**validate_uaid**(*uaid*)
Validates a UAID a tuple indicating if its valid and the original uaid, or a new uaid if its invalid

autopush.utils.**generate_hash**(*key*, *payload*)
Generate a HMAC for the uaid using the secret

Returns HMAC hash and the nonce used as a tuple (nonce, hash).

autopush.websocket

Websocket Protocol handler and HTTP Endpoints for Connection Node

Private HTTP Endpoints

These HTTP endpoints are only for communication from endpoint nodes and must not be publicly exposed.

PUT /push/(**uuid:** *uaid*)
Send a notification to a connected client with the given *uaid*.

Status Codes

- 200 OK – Client is connected and delivery will be attempted.
- 404 Not Found – Client is not connected to this node.

- 503 Service Unavailable – Client is connected, but currently busy.

PUT /notif/ (**uuid:** *uaid*)
Trigger a stored notification check for a connected client.

Status Codes

- 200 OK – Client is connected, and has started checking.
- 202 Accepted – Client is connected but busy, will check notifications when not busy.
- 404 Not Found – Client is not connected to this node.

DELETE /notif/ (**uuid:** *uaid*) **/**
int: *connected_at* Immediately drop a client of this *uaid* if its connection time matches the *connected_at* provided.

Websocket Protocol

class `autopush.websocket.`**PushServerProtocol**
Main Websocket Connection Protocol

parent_class
alias of `WebSocketServerProtocol`

randrange (*start*, *stop=None*, *step=1*, *_int=<type 'int'>*, *_maxwidth=9007199254740992L*)
Choose a random item from range(start, stop[, step]).

This fixes the problem with randint() which includes the endpoint; in Python this is usually not what you want.

deferToThread (*func*, **args*, ***kwargs*)
deferToThread helper that tracks defers outstanding

deferToLater (*when*, *func*, **args*, ***kwargs*)
deferToLater helper that tracks defers outstanding

force_retry (*func*, **args*, ***kwargs*)
Forcefully retry a function in a thread until it doesn't error

Note that this does not use `self.deferToThread`, so this will continue to retry even if the client drops.

base_tags
Property that uses None if there's no tags due to a DataDog library bug

log_failure (*failure*, ***kwargs*)
Log a twisted failure out through twisted's log.failure

paused
Indicates if we are paused for output production or not

_sendAutoPing (**args*, ***kwargs*)
Override for sanity checking during auto-ping interval

sendClose (**args*, ***kwargs*)
Override to add tracker that ensures the connection is truly torn down

nukeConnection (**args*, ***kwargs*)
Aggressive connection shutdown using abortConnection if onClose still hadn't run by this point

onConnect (**args*, ***kwargs*)
autobahn onConnect handler for when a connection has started

processHandshake(**args*, ***kwargs*)
Disable host port checking on nonstandard ports since some clients are buggy and don't provide it

onMessage(**args*, ***kwargs*)
autobahn onMessage processor for incoming messages

timeoutConnection()
Idle timer fired.

onAutoPingTimeout()
Override to track that this shut-down is from a ping timeout

onClose(**args*, ***kwargs*)
autobahn onClose handler for shutting down the connection and any outstanding deferreds related to this connection

cleanUp(*wasClean*, *code*, *reason*)
Thorough clean-up method to cancel all remaining deferreds, and send connection metrics in

_save_webpush_notif(*notif*)
Save a direct_update webpush style notification

_save_simple_notif(*channel_id*, *version*)
Save a simplepush notification

_lookup_node(*results*)
Looks up the node to send a notify for it to check storage if connected

_trap_uaid_not_found(*fail*)
Traps UAID not found error

_notify_node(*result*)
Checks the result of lookup node to send the notify if the client is connected elsewhere now

returnError(*messageType*, *reason*, *statusCode*, *close=True*, *message=''*, *url='http://autopush.readthedocs.io/en/latest/api/websocket.html#private-http-endpoint'*)
Return an error to a client, and optionally shut down the connection safely

err_overload(*failure*, *message_type*, *disconnect=True*)
Handle database overloads and errors

If `disconnect` is False, the an overload error is returned and the client is not disconnected.

Otherwise, pause producing to cease incoming notifications while we wait a random interval up to 8 seconds before closing down the connection. Most clients wait up to 10 seconds for a command, but this is not a guarantee, so rather than never reply, we still shut the connection down.

Parameters **disconnect** – Whether the client should be disconnected or not.

err_finish_overload(*message_type*)
Close the connection down and resume consuming input after the random interval from a db overload

sendJSON(*body*)
Send a Python dict as a JSON string in a websocket message

process_hello(*data*)
Process a hello message

_register_user(*existing_user=True*)
Register a returning or new user

_verify_user_record()
Verify a user record is valid

Returns a record that is ready for registering in the database if the user record was found.

Return type `Item` or None

err_hello(*failure*)
errBack for hello failures

_check_other_nodes(*result*, *url='http://autopush.readthedocs.io/en/latest/api/websocket.html#private-http-endpoint'*)
callback to check other nodes for clients and send them a delete as needed

finish_hello(*previous*)
callback for successful hello message, that sends hello reply

process_notifications()
Run a notification check against storage

webpush_fetch()
Helper to return an appropriate function to fetch messages

error_notifications(*fail*)
errBack for notification check failing

error_notification_overload(*fail*)
errBack for provisioned errors during notification check

error_message_overload(*fail*)
errBack for handling excessive messages per UAID

finish_notifications(*notifs*)
callback for processing notifications from storage

finish_webpush_notifications(*result*)
WebPush notification processor

_rotate_message_table()
Function to fire off a message table copy of channels + update the router current_month entry

_monthly_transition()
Transition the client to use a new message month

Utilized to migrate a users channels to a new message month and update the router record reflecting the proper month.

This is a blocking function that does *not* run on the event loop.

_finish_monthly_transition(*result*)
Mark the client as successfully transitioned and resume

error_monthly_rotation_overload(*fail*)
Capture overload on monthly table rotation attempt

If a provision exdeeded error hits while attempting monthly table rotation, schedule it all over and re-scan the messages. Normal websocket client flow is returned in the meantime.

_send_ping()
Helper for ping sending that tracks when the ping was sent

process_ping()
Ping Handling

Clients in the wild have a bug that lowers their ping interval to 0. It will never increase for them, as there is no way to remedy this without causing the client to use drastically more battery/data-usage we send them a code 4774 close to signify that they should stop until network change.

No other client should ping more than once per minute, or we tell them to go away.

process_register(*data*)
Process a register message

error_register(*fail*)
errBack handler for registering to fail

finish_register(*endpoint*, *chid*)
callback for successful endpoint creation, sends register reply

process_unregister(*data*)
Process an unregister message

ack_update(*update*)
Helper function for tracking ack'd updates

Returns either None, if no delete_notification call is needed, or a deferred for the delete_notification call if it was needed.

_handle_webpush_ack(*chid*, *version*, *code*)
Handle clearing out a webpush ack

_handle_webpush_update_remove(*result*, *chid*, *notif*)
Handle clearing out the updates_sent

It's possible the client may leave before this runs, so this is wrapped in a try/except in case the tear-down of self has started.

_handle_simple_ack(*chid*, *version*, *code*)
Handle clearing out a simple ack

process_ack(*data*)
Process an ack message, delete notifications from storage if needed

process_nack(*data*)
Process a nack message and log its contents

check_missed_notifications(*results*, *resume=False*)
Check to see if notifications were missed

bad_message(*typ*, *message=None*, *url='http://autopush.readthedocs.io/en/latest/api/websocket.html#private-http-endpoint'*)
Error helper for sending a 401 status back

_newer_notification_sent(*channel_id*, *version*)
Returns whether a newer channel_id/version has already been sent

send_notification(*update*)
Utility function for external use

This function is called by the HTTP handler to deliver an incoming update notification from an endpoint.

HTTP Handlers

class autopush.websocket.**RouterHandler**(*application*, *request*, ***kwargs*)
Router Handler

Handles routing a notification to a connected client from an endpoint.

put(*uaid*)
HTTP Put

Attempt delivery of a notification to a connected client.

class autopush.websocket.**NotificationHandler**(*application*, *request*, ***kwargs*)

put(*uaid*, **args*)
HTTP Put

Notify a connected client that it should check storage for new notifications.

delete(*uaid*, *connected_at*)
HTTP Delete

Drop a connected client as the client has connected to a new node.

Utility Functions

autopush.websocket.**ms_time**()
Return current time.time call as ms and a Python int

autopush.websocket.**log_exception**(*func*)
Exception Logger Decorator for protocol methods

autopush.jwt

class autopush.jwt.**VerifyJWT**
Minimally verify a Vapid JWT object.

Why hand roll? Most python JWT libraries either use a python elliptic curve library directly, or call one that does, or is abandoned, or a dozen other reasons.

After spending half a day looking for reasonable replacements, I decided to just write the functions we need directly.

THIS IS NOT A FULL JWT REPLACEMENT.

static extract_signature(*auth*)
Fix the JWT auth token.

The JWA spec defines the signature to be a pair of 32octet encoded longs. The *ecdsa* library signs using a raw, 32octet pair of values (s, r). Cryptography, which uses OpenSSL, uses a DER sequence of (s, r). This function converts the raw ecdsa to DER.

Parameters **auth** (*str*) – A JWT authorization token.

:return tuple containing the signature material and signature

static extract_assertion(*token*)
Extract the assertion dictionary from the passed token. This does NOT do validation.

Parameters **token** – Partial or full VAPID auth token

:return dict of the VAPID claims

static validate_and_extract_assertion(*token*, *key*)
Decode a web token into a assertion dictionary.

This attempts to rectify both ecdsa and openssl generated signatures. We use the built-in cryptography library since it wraps libssl and is faster than the python only approach.

Parameters

- **token** (*str*) – VAPID auth token
- **key** (*str or bitarray*) – bitarray containing public key

:return dict of the VAPID claims

:raise InvalidSignature

CHAPTER 5

Changelog

CHAPTER 6

Bugs/Support

Bugs should be reported on the autopush github issue tracker.

The developers of `autopush` can frequently be found on the Mozilla IRC network (irc.mozilla.org) in the #push channel.

CHAPTER 7

autopush Endpoints

autopush is automatically deployed from master to a dev environment for testing, a stage environment for tagged releases, and the production environment used by Firefox/FirefoxOS.

dev

- Websocket: wss://autopush.dev.mozaws.net/
- Endpoint: https://updates-autopush.dev.mozaws.net/

stage

- Websocket: wss://autopush.stage.mozaws.net/
- Endpoint: https://updates-autopush.stage.mozaws.net/

production

- Websocket: wss://push.services.mozilla.com/
- Endpoint: https://updates.push.services.mozilla.com/

CHAPTER 8

Reference

- genindex
- modindex
- *Glossary*

Glossary

AppServer A third-party Application Server that delivers notifications to client applications via Push.

Bridging Using a third party or proprietary network in order to deliver Push notifications to an App. This may be preferred for mobile devices where such a network may improve battery life or other reasons.

Channel A unique route between an *AppServer* and the Application. May also be referred to as *Subscription*

CHID The Channel Subscription ID. Push assigns each subscription (or channel) a unique identifier.

Message-ID A unique message ID. Each message for a given subscription is given a unique identifier that is returned to the *AppServer* in the `Location` header.

Notification A message sent to an endpoint node intended for delivery to a HTTP endpoint. Autopush stores these in the message tables.

Router Type Every *UAID* that connects has a router type. This indicates the type of routing to use when dispatching notifications. For most clients, this value will be `webpush`. Older Firefox OS clients use `simplepush` and clients using *Bridging* it will be one of `gcm`, `fcm`, or `apns`.

Subscription A unique route between an *AppServer* and the Application. May also be referred to as a *Channel*

UAID The Push User Agent Registration ID. Push assigns each remote recipient (Firefox client) a unique identifier. These may occasionally be reset by the Push Service or the client.

WebPush An IETF standard for communication between Push Services, the clients, and application servers.

See: https://datatracker.ietf.org/doc/draft-ietf-webpush-protocol/

CHAPTER 9

License

`autopush` is offered under the Mozilla Public License 2.0.

HTTP Routing Table

/m

```
DELETE /m/{message_id}, 6
```

/notif

```
PUT /notif/(uuid:uaid), 46
DELETE /notif/(uuid:uaid)/(int:connected_at),
       46
```

/push

```
PUT /push/(uuid:uaid), 45
```

/v1

```
GET /v1/{type}/{app_id}/registration/{UAID}/,
       10
POST /v1/{type}/{app_id}/registration,
       8
POST /v1/{type}/{app_id}/registration/{uaid}/subscription,
       9
PUT /v1/{type}/{app_id}/registration/{uaid},
       8
DELETE /v1/{type}/{app_id}/registration/{UAID}/subscription/{CHID},
       10
DELETE /v1/{type}/{app_id}/registration/{uaid},
       9
```

/{push_endpoint}

```
POST {push_endpoint}, 5
```

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

R

S

T

U

V

W