hookbox Documentation

Release 0.3.4

Michael Carter

February 23, 2013

CONTENTS

Contents:

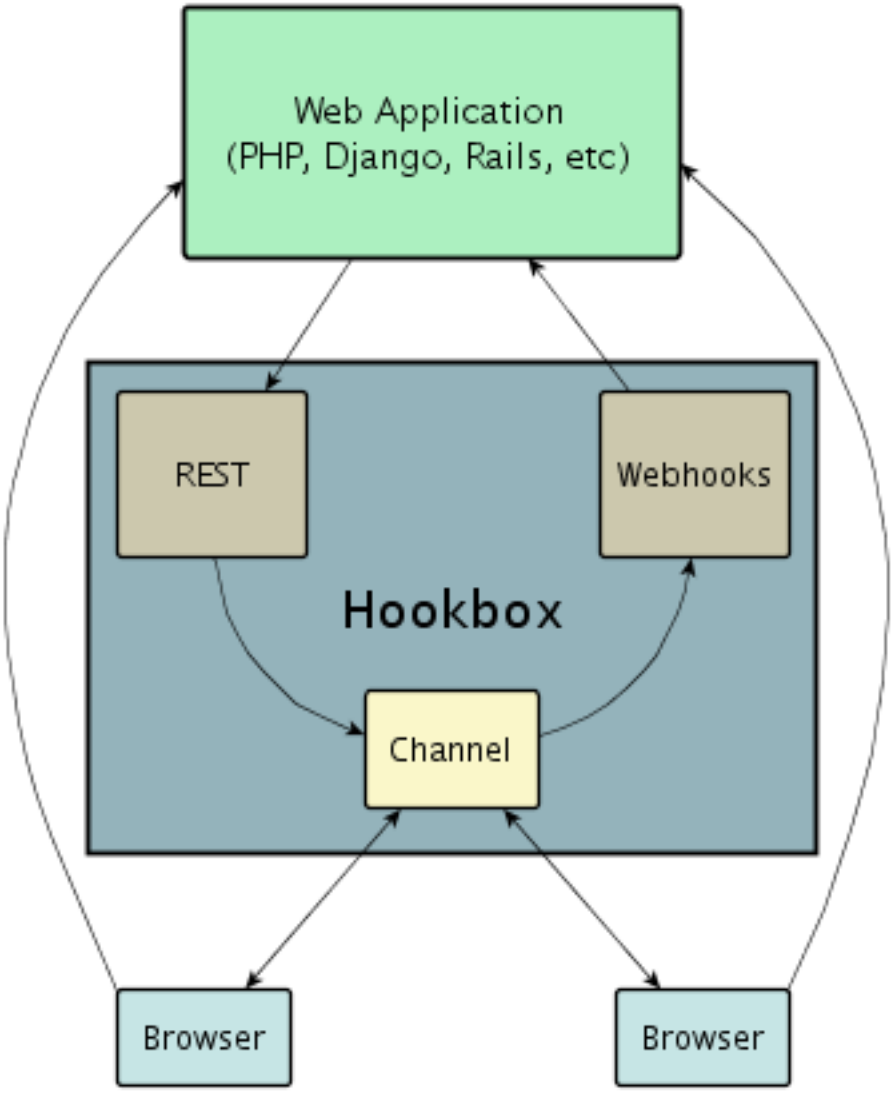
CHAPTER

ONE

INTRODUCTION

1.1 Overview

Hookbox's purpose is to ease the development of real-time web applications, with an emphasis on tight integration with existing web technology. Put simply, Hookbox is a web-enabled message queue. Browers may directly connect to Hookbox, subscribe to named channels, and publish and receive messages on those channels in real-time. An external application (typically the web application itself) may also publish messages to channels by means of the Hookbox REST interface. All authentication and authorization is performed by an external web application via designated "webhook" callbacks.



Any time a user connects or operates on a channel, (subscribe, publish, unsubscribe) Hookbox makes an http request to the web application for authorization for the action. Once subscribed to a channel, the user's browser will receive real-time events that originate either in another browser via the javascript api, or from the web application via the REST api.

They key insight is that all application development with hookbox Happens either in javascript, or in the native language of the web application itself (e.g. PHP.)

1.2 Terminology

Throughout this documentation, we throw various terms around like you should know what we're talking about. This is probably not always the case, so this section includes a list of terms that have caused trouble for previous readers.

- `Application`, `Web Application`, `Web app`: An HTTP based application; commonly written with PHP, Django, Ruby on Rails, ASP.NET, and Java servlets.
- `Browser`, `Client`: Firefox, IE, Chrome, Safari, Opera, or some variant.

- `Webhook`, `HTTP callback`: A HTTP request made from Hookbox to the Web app when various events occur on the Hookbox server.

1.3 Common Patterns

Hookbox is built around the concept of named *Channels* which are extremely flexible. These channels can be configured by the web application to provide many features out of the box. Here are a few typical patterns:

1.3.1 Real-time graph (time series)

When a user connects to a web page with a real-time, time series graph, that user needs access to the last N data points on the graph. After that, the user needs updates on an interval. Lets say that we want to build a graph that shows a new data point every second, and always shows the last 30 seconds of data. Firstly, we enable a history on the channel with:

```
"history_size": 30
```

This will cause the channel to send any new subscribers the last 30 data points immediately when they subscribe.

Next we need to put data into the channel on an interval. There are a number of ways we could do this, for instance we have access to the REST API so we could set a cron job that calls http://HOOKBOX_HOST:HOOKBOX_PORT/rest/publish every second. But this method is clumsy and imposes additional deployment constraints on our application. Instead, we can use the channel polling feature to populate our channel with data from a remote url every second. Assuming we had an application hosted at http://example.org/cpu_usage which returns a single number representing our target data, we can set polling up like so:

```
"polling": {
    "url": "http://example.org/cpu_usage",
    "interval": 1.0,
    "mode": "simple"
}
```

Now our channel will automatically poll our application for new data points every second, and then rebroadcast that too all subscribers.

Putting it all together, our actual implementation would consist of just a few pieces.

1. A create_channel callback which returns the polling and history_size options. This is just a url in the web application which hookbox will issue and HTTP request to when someone first attempts to subscribe to the channel and it needs to be created.
2. A connect callback which returns `[true, { "name": "$username" } ]`. This is again a url in the web application that hookbox will call when someon tries to connect. The value of $username would just be a random string in this case.
3. A subscribe callback which returns `[true, {}]` so that users can subscribe to the channel
4. A publish callback which returns `[false, {}]` so that users cannot publish to the channel
5. an html with javascript that connects to hookbox, graphs the initial history, and graphs all new data points as they come in.

If you're interested in seeing a completed version of this type of application then take a look at the PHP Real-time Time Series Graph on github.

1.3.2 Chat Room (with presence)

A chat room should contain a history, and presence information (We should see who is in the room, and receives notifications when users join or leave.) Also, its nice to have the server send your own chat messages back to you so you can be sure they went through, so we will turn on reflection. All we need to do is implement a create_channel callback that sets this up by returning:

```
[ true, { "history_size": 20, "presenceful": true, "reflective": true } ]
```

In some cases it is nice to persist all chat messages, joins, and leaves, so our web application needs to write these events to the database within the `publish`, `subscribe`, and `unsubscribe` callbacks.

If we have that information stored in a database, then we can fetch it out to pre-populate the history of the channel from the last conversation by returning the a "history" setting as well. Typically a history setting might look like this:

```
"history": [
    [
        "UNSUBSCRIBE",
        {
            "user": "mcarter"
        }
    ],
    [
        "PUBLISH",
        {
            "payload": "Anyone there?",
            "user": "mcarter"
        }
    ],
    [
        "SUBSCRIBE",
        {
            "user": "mcarter"
        }
    ]
]
```

1.3.3 Card Game

TODO

1.4 Installation

Hookbox is written in python and depends on setuptools for installation. The fastest way to install hookbox is to type:

```
# easy_install hookbox
```

If you are missing python or setuptools, please refer to the following links:

- install python
- install setuptools

To confirm your installation succeeded, type:

```
# hookbox --help
```

1.5 Github

The development version of Hookbox is located on github:

- http://github.com/hookbox/hookbox

You can get a copy of the latest source by cloning the repository:

```
# git clone git://github.com/hookbox/hookbox.git
```

To install hookbox from source, ensure you have python and setuptools, then run:

```
# cd hookbox/hookbox
# python setup.py install
```

CHAPTER

TWO

TUTORIAL

CHAPTER

THREE

CHANNELS

3.1 Overview

Hookbox is built around the concept channels which can be used as an abstraction for routing real-time communication between browers and your web application. These channels have many options, making them suitable for a multitude of basic types of applications. You need to consider the features needed by the real-time portion of your application, and adjust the channel options appropriately. You can read about some *Common Patterns* to find a good starting point for your application.

Ultimately, your web application has complete control over these channels. A user may perform three actions on a channel, subscribing, publishing, or unsubscribing. Whenever a user attempts to perform an action on a channel, Hookbox will make a *Webhook* call back to your application to obtain permission for the action by that user, or simply to notify your web application that the action took place.

The application may itself perform any of these actions on a channel by using the *REST API*; it may perform any action on behalf of any user, and it can even publish with arbitrary usernames. The application may additionally use the REST API to alter a channel's state or options at any time.

Finally, the web application may sometimes perform actions on channels on behalf of a particular user when that user causes a webhook callback. For instance, a user might subscribe to the channel 'foo', which would result in a `subscribe` webhook to be issued. The web application may respond with the `auto_subscribe` directive in order to subscribe the user to another channel, such as 'bar'.

You can think of channels as three parts:

1. Unique name
2. List of actively connected/subscribed users
3. History/state

And you can think of the operations you can perform on a channel in three categories:

1. Putting data into a channel
2. Getting data out of a channel
3. Altering channel subscriptions

3.2 Getting Data out of a Channel

The most common way to get data out of a channel is to simply subscribe with the *Javascript API*. This method allows javascript code to attach a publish callback which will be invoked whenever a new message is published to the channel.

3.3 Putting Data into Channels

3.4 Channel Properties

- `history_size`: the maximum number of entries in the channel history.
- `history_duration`: the maximum seconds of duration of life in the channel history.
- `history`: A list of events that have previously occurred on this channel. They may be Subscribe, Unsubscribe, or Publish events.
- `history_publish_only`: A Boolean to indicate that only Publish events are in the history.
- `name`: The name of the channel.
- `presenceful`: A Boolean indicating whether presence information is shared with channel subscribers.
- `moderated`: If true, any action will cause a Webhook callback.
- `moderated_publish`: If true, Publish will cause a Webhook callback.
- `moderated_subscribe`: If true, Subscribe will cause a Webhook callback.
- `moderated_unsubscribe`: If true, Unsubscribe will cause a Webhook callback.
- `reflective`: messages sent to this channel will also be sent back to the sender
- `server_presenceful`: Needs to know about server presence
- `server_user_presence`: A list of ,
- `anonymous`: messages sent to this channel will not contain user information
- `polling`: A dictionary containing the values `mode`: “”, `interval`: 5.0, `url`: “”, `form`: {}, `originator`: “”
- `state`: {}
- TODO: etc.

CHAPTER

FOUR

JAVASCRIPT API

The Hookbox javascript api is contained completely in the hookbox.js file that comes with the Hookbox daemon. The file is served by hookbox at http://HOOKBOX_HOST:HOOKBOX_PORT/static/hookbox.js, but you may serve the file from any location and it will still work.

4.1 Connecting

To connect to hookbox you need to provide the url of the hookbox server, with a "/csp" as the path. For example, if hookbox server is running on host "localhost" and port 8001, (the default) then the to connect:

```
var conn = hookbox.connect('http://localhost:8001/csp')
```

Once the connection has been successfully established, the `onopen` callback will be called. You can attach your own callback to catch:

```
conn.onOpen = function() { alert("connection established!"); }
```

If there was an error when connecting the `onError` callback will be invoked:

```
conn.onError = function(err) { alert("connection failed: " + err.msg); }
```

4.2 Disconnecting

To disconnect from hookbox use the `conn.disconnect` method:

```
conn.disconnect();
```

NOTE: This method will often not result in a successful disconnect if called in the unload handler for the web page, in which case the user won't be disconnected until they timeout (after about 60 seconds)

4.3 Subscribing to Channels

As soon as you have a connection object after calling hookbox.connect, you are free to make calls to conn.subscribe, even before the connection is established. These calls will be buffered until after the onOpen event is fired.

To subscribe to a channel use the conn.subscribe function:

```
conn.subscribe("my_channel_name");
```

There is no returned object when calling conn.subscribe; rather, a subscription object is passed to you through the onSubscribed callback once the subscription is successful.

```
var subscription = null;
conn.onSubscribed = function(channelName, _subscription, args) {
  subscription = _subscription;
}
```

Its important to understand that the `onSubscribed` callback can be called even if you've never made a call to subscribe. This might be because the web application decided to `auto_subscribe` you to some channel, or it could be because the user is already logged in and subscribed to multiple channels, though in a different browser window or tab. If the subscribe call is made successfully in another tab, then this tab's Hookbox connection object will also issue an `onSubscribed` callback.

4.4 Interacting with Channels

Once you have a subsription object, you are able to inspect the channel's attributes, publish to the channel, and receive publishes from other subscribers in the channel.

4.4.1 Channel Attributes

If the channel is set to have `history_size` > 0, then you will have access to history information for that channel:

```
>>> subscription.history
[["PUBLISH", Object { user="mcarter", payload="greetings!"}], ["SUBSCRIBE", Object { user="mcarter" }
```

All attributes are read only. The complete list:

- `historySize`: the length of the history for the channel.
- `history`: a list of the last N elements where N is the `history_size` attribute of the channel
- `state`: arbitrary (json) data set on the channel by the web application. This attribute updates automatically when the web application changes it, and an onState callback is issued on the subscription.
- `presenceful`: boolean that signifies rather this channel relays presence information
- `presence`: a list of users subscribed to the channel. This is always empty if `presenceful` is false.
- `reflective`: boolean signifying if this channel reflects publish frames back to the connection that orignated them.

4.4.2 Presence Information

Note in the above example that one of the frames in the history is `SUBSCRIBE`. The channel will only relay subscribe and unsubscribe frames to the browser if `presenceful` = true is set on the channel by the web application. If it is set, then the subscription object will provide access to a list of users currently subscribed to this channel:

```
>>> subscription.presence
[ "mgh", "mcarter", "desmaj" ]
```

Whenever a user subscribes or unsubscribes from the channel you will receive an `onSubscribe` or `onUnsubscribe` callback from the subscription, and the presence attribute will be updated.

```
subscription.onSubscribe = function(frame) {
  // the user is now in our presence list
  assertTrue(subscription.presence.indexOf(frame.user) != -1);
  alert("user: " + frame.user + " has subscribed!");
}

subscription.onUnsubscribe = function(frame) {
  // the user is no longer in our presence list
  assertTrue(subscription.presence.indexOf(frame.user) == -1);
  alert("user: " + frame.user + " has unsubscribed!");
}
```

4.4.3 Publishing

Perhaps the most important part of interacting with channels is publishing data receiving published data. You may publish data by calling the `subscription.publish` method:

```
subscription.publish(42);
subscription.publish({foo: "bar"});
subscription.publish(null);
subscription.publish([1,2,3, {a: [4,5,6] });
```

As you can see, any native javascript object that can be transported as `JSON` is legal.

Whenever data is published to the channel, the onPublish callback on the subscription will be called. If the `reflective` attribute is set on the channel by the web application, then your own calls to publish will cause an onPublish callback as well.

```
subscription.onPublish = function(frame) {
  alert(frame.user + " said: " + frame.payload);
}
```

Remember, frame.payload can be any javascript object that can be represented as `JSON`.

4.4.4 State

It sometimes makes sense for the web application to stash some additional state information on the channel either by setting it in a webhook callback response, or using the rest api. In javascript, the subscription object maintains the `state` attribute and issues onState callbacks whenever this attribute is modified. The state cannot be modified by the client; it is unidirectional only. The `state` attribute is always a valid json object {}.

```
subscription.onState = function(frame) {
    var updates = frame.updates; // object with the new keys/values and
                                 // modified keys/values

    var deletes = frame.deletes; // list containing all deleted keys.

    // No need to compute the state from the updates and deletes, its done
    // for you and stored on subscription.state
    alert('the name state is: ' + JSON.stringify(subscription.state));
}
```

4.4.5 Unsubscribing

You can use the javascript client to request that the user be unsubscribed from a channel with the `subscription.cancel` method. When the subscription has been successfully canceled, the `conn.onUnsubscribed` will be issued. Keep in mind that the web app may override this request and not allow the user to be unsubscribed and so the onUnsubscribed callback will not be issued.

```
subscription = conn.subscribe('foo.bar.baz')
    ...
    conn.onUnsubscribed = function(subscription, frame) {
            alert('successfully unsubscribed from: ' + subscription.channelName);
    }
subscription.cancel();
```

CHAPTER

FIVE

WEBHOOKS

5.1 publish

Send a message to all users subscribed to a channel.

Webhook Form Variables:

- `channel_name`: The name of the channel the message is being published to.
- `payload`: The json payload to send to all users subscribed to the channel.

Webhook post includes sender cookies.

Returns json:

```
[ success (boolean) , details (object) ]
```

Optional Webhook return details:

- `override_payload`: A new payload that will be published instead of the original payload.
- `only_to_sender`: If true, the message will only be published to the sender instead of all the users subscribed the channel.
- `error`: If success is false, error text to return to sender.

Example:

Client Calls:

```
connection.publish("channel-1", { title: "a message", body: "some text" });
```

Webhook Called With:

```
{ channel_name: "channel-1", payload: { title: "a message", body: "some text" } }
```

Webhook replies:

```
[ true, { } ]
```

And the following frame is published all subscribers to the channel 'channel-1':

```
{ channel_name: "channel-1", "payload": { title: "a message", body: "some text" } }
```

5.2 message

Send a private message to a user.

Webhook Form Variables:

- `sender`: The user name of the sending user.
- `recipient`: The user name of the receiving user.
- `recipient_exists`: True if the recipient name is that of a connected user, false otherwise.
- `payload`: The json payload to send to the receiving user.

Webhook post includes sender cookies.

Returns json:

```
[ success (boolean) , details (object) ]
```

Optional Webhook return details:

- `override_payload`: A new payload that will be sent instead of the original payload.
- `override_recipient_name`: The name of a user to send the message to instead of the original reciepient.

Example:

Client Calls:

```
connection.message("mcarter", { title: "a message", body: "some text" });
```

Webhook Called With:

```
{ sender: "some_user", recipient: "mcarter", payload: { title: "a message", body: "some text" } }
```

Webhook replies:

```
[ true, { override_payload: { title: "a new title", body: "some text" } } ]
```

And the following frame is published to the user 'mcarter':

```
{ sender: "some_user", recipient: "mcarter", "payload": { title: "a new title", body: "some text" } }
```

CHAPTER

SIX

WEB/HTTP INTERFACE

6.1 publish

Publish a message to a channel.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.
- `channel_name`: The target channel.
- `payload`: The json payload to publish

Optional Form Variables:

- `originator`: The name of the user who will appear to do the publish

Returns json:

```
[ success (boolean) , details (object) ]
```

Example:

Client Requests URL:

```
/web/publish?security_token=yo&channel_name=testing&payload=[1, 2, "foo"]&originator=dictator
```

Server Replies:

```
[ true, {} ]
```

And the following frame is published to channel 'testing':

```
{ "user": dictator, "payload": [1, 2, "foo"] }
```

6.2 subscribe

Add a user to a channel.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.
- `channel_name`: The target channel.
- `name`: The name of the target user.

Returns json:

```
[ success (boolean) , details (object) ]
```

Example:

Client Requests URL:

```
/web/subscribe?security_token=yo&channel_name=testing&user=mcarter
```

Server Replies:

```
[ true, {} ]
```

And the user "mcarter" is subscribed to the channel "testing".

6.3 unsubscribe

Remove a user from a channel.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.
- `channel_name`: The target channel.
- `name`: The name of the target user.

Returns json:

```
[ success (boolean) , details (object) ]
```

Example:

Client Requests URL:

```
/web/unsubscribe?security_token=yo&channel_name=testing&user=mcarter
```

Server Replies:

```
[ true, {} ]
```

And the user "mcarter" is unsubscribed from the channel "testing".

6.4 message

Publish a message to a user.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.
- `sender_name`: The user name of the message sender.
- `recipient_name`: The user name of the message recipient.
- `payload`: The json payload to send

Returns json:

```
[ success (boolean) , details (object) ]
```

Example:

Client Requests URL:

```
/web/message?security_token=yo&sender_name=bob&recipient_name=joe&payload=[1, 2, "foo"]
```

Server Replies:

```
[ true, {} ]
```

And the following message frame is sent to user 'joe':

```
{ "sender": "bob", "recipient": "joe", "payload": [1, 2, "foo"] }
```

6.5 get_channel_info

Returns all settings and attributes of a channel.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.
- `channel_name`: The target channel.

Returns json:

[success (boolean) , details (object)]

Example:

Client Requests URL:

```
/web/get_channel_info?security_token=yo&channel_name=testing
```

Server Replies:

```
[
    true,
    {
        "name": "testing",
        "options": {
            "anonymous": false,
            "history": [
                [
                    "SUBSCRIBE",
                    {
                        "user": "mcarter"
                    }
                ],
                [
                    "PUBLISH",
                    {
                        "payload": "good day",
                        "user": "mcarter"
                    }
                ],
                [
                    "PUBLISH",
```

```
                    {
                        "payload": "was gibt es?",
                        "user": "mcarter"
                    }
                ]
            ],
            "history_duration": 0,
            "history_size": 5,
            "moderated": false,
            "moderated_publish": true,
            "moderated_subscribe": true,
            "moderated_unsubscribe": true,
            "polling": {
                "form": {},
                "interval": 5,
                "mode": "",
                "originator": "",
                "url": ""
            },
            "presenceful": true,
            "reflective": true
        },
        "subscribers": [
            "mcarter"
        ]
    }
]
```


6.6 set_channel_options

Set the options on a channel.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.
- `channel_name`: The target channel.

Optional Form Variables:

- `anonymous`: json boolean
- `history`: json list in the proper history format
- `history_duration`: json integer
- `history_size`: json integer
- `moderated`: json boolean
- `moderated_publish`: json boolean
- `moderated_subscribe`: json boolean
- `moderated_unsubscribe`: json boolean
- `polling`: json object in the proper polling format
- `presenceful`: json boolean
- `reflective`: json boolean

- `state`: json object

Example:

Client Requests URL:

```
/web/set_channel_options?security_token=yo&channel_name=testing&history_size=2&presenceful=true
```

Server Replies:

```
[ true, {} ]
```

The `history_size` of the channel is now *2*, and `presenceful` is *false*.

6.7 create_channel

TODO

6.8 destroy_channel

TODO

6.9 state_set_key

Sets a key in a channel's state object. If the key already exists it is replaced, and if not it is created.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.
- `channel_name`: The target channel.

Optional Form Variables:

- `key`: The target key in the state
- `val`: any valid json structure; it will be the new value of the given key on the state

Example:

Client Requests URL:

```
/web/state_set_key?security_token=yo&channel_name=testing&key=score&val={ "mcarter": 5, "desmaj": 11
```

Server Replies:

```
[ true, {} ]
```

The `state` of the channel now contains the key "testing" with the value { "mcarter": 5, "desmaj": 11 }. An onState javascript callback will be issued to all subscribers; They will be able to access subscription.state.score.mcarter and will see the value 5.

6.10 state_delete_key

Removes a key from the state of a channel. If the key doesn't exist then nothing happens.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.
- `channel_name`: The target channel.

Optional Form Variables:

- `key`: The target key in the state to delete

Example:

Client Requests URL:

```
/web/state_delete_key?security_token=yo&channel_name=testing&key=score
```

Server Replies:

```
[ true, {} ]
```

The `state` of the channel no longer contains the key "score". An onState callback will be issued to all subscribers.

6.11 set_config

Update certain configuration parameters (mostly webhook related options) immediately without restarting hookbox.

Required Form variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.

Optional Form Variables:

- `cbhost`: json string
- `cbport`: json integer
- `cbpath`: json string
- `cb_connect`: json string
- `cb_disconnect`: json string
- `cb_create_channel`: json string
- `cb_destroy_channel`: json string
- `cb_subscribe`: json string
- `cb_unsubscribe`: json string
- `cb_publish`: json string
- `cb_single_url`: json string
- `admin_password`: json string
- `webhook_secret`: json string
- `api_security_token`: json string

Example:

Client Requests URL:

```
/web/state_delete_key?security_token=yo&cbhost="1.2.3.4&cbport=80
```

Server Replies:

```
[ true, {} ]
```

The callback host is now set to `1.2.3.4` and the port is now `80`.

6.12 get_user_info

Returns all settings and attributes of a user.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.
- `user_name`: The target user.

Returns json:

[success (boolean) , details (object)]

Example:

Client Requests URL:

```
/web/get_user_info?security_token=yo&user_name=mcarter
```

Server Replies:

```
[
    true,
    {
        "channels": [
            "testing"
        ],
        "connections": [
            "467412414c294f1a9d1759ace01455d9"
        ],
        "name": "mcarter",
        "options": {
            "reflective": true,
            "moderated_message": true,
            "per_connection_subscriptions": false
        }
    }
]
```

6.13 set_user_options

Set the options for a user.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.

- `user_name`: The target user.

Optional Form Variables:

- `reflective`: json boolean - if true, private messages sent by this user will also be sent back to the user
- `moderated_message`: json boolean - if true, private messages sent by this user will call the message webhook
- `per_connection_subscriptions`: json boolean - if true, only the user connection (or connections) that sends a subscribe frame will be subscribed to the specified channel. Otherwise, all of a user's connections will share channel subscriptions established by any of the connections.

Example:

Client Requests URL:

```
/web/set_user_options?security_token=yo&user_name=mcarter&reflective=false
```

Server Replies:

```
[ true, {} ]
```

The `reflective` of the user is now *false*.

6.14 get_server_info

Returns all current users and connections of the server.

Required Form Variables:

- `security_token`: The password specified in the config as `-r` or `--api-security-token`.

Returns json:

[success (boolean) , details (object)]

Example:

Client Requests URL:

```
/web/get_server_info?security_token=yo
```

Server Replies:

```
[
    true,
    {
        "channels": [
            "testing",
            "testing2"
        ],
        "connections": [
            "467412414c294f1a9d1759ace01455d9",
            "759ace01455d9467412414c294f1a9d1",
            "14c294f1a9d1759ace01455d94674124"
        ]
    }
]
```

CHAPTER

SEVEN

JSON REST INTERFACE

TODO

CHAPTER

EIGHT

CONFIGURATION

As of version 0.2, hookbox is completely configurable via command line options. While developing your application, you will want to create a startup script that contains all of the appropriate settings.

A typical hookbox start command looks like this:

```
# hookbox -a myadminpassword -r myapitoken -s mycallbacksecret
```

8.1 Basic Options

8.1.1 Port (-p, –port)

The port hookbox binds to is specified by -p PORT or –port=PORT; the default is 8001.

8.1.2 Interface (-i, –interface)

The interface hookbox binds to is specified by -i INTERFACE or –interface=INTERFACE; the default is “0.0.0.0”

8.2 Webhook Callback Options

8.2.1 Callback Port (–cbport)

The port of the web application which will handle webhook callbacks is specified by –cbport=PORT; the default is 80.

8.2.2 Callback Hostname (–cbhost)

The hostname of the web application which will handle webhook callbacks is specified by –cbhost=HOSTNAME; the default is “localhost”.

8.2.3 Callback Path Prefix (–cbpath)

All callbacks will be prefixed with the value specified by –cbpath=PATH_PREFIX; the default is “/hookbox”.

8.2.4 Callback Secret Token (-s, –webhook-secret)

If a secret token is provided, all callbacks with include that token value as the form variable "secret"; this is useful for blocking unauthorized requests to the callback urls. The secret is specified by -s SECRET or –webhook-secret=SECRET; the default is null (no secret.)

8.2.5 Callback Hookbox Version (–cbsendhookboxversion)

Send hookbox version info to webhook callbacks using X-Hookbox-Version header.

8.2.6 Callback via Https (–cbhttps)

Use https (instead of http) to make callbacks.

8.2.7 Callback Add Trailing slash (–cbtrailingslash)

Append a trailing slash to the callback URL if there is none.

8.3 Extended Callback Options

These options are typically left as default, except in cases where its helpful to point all callbacks at a single url, for instance a single PHP script.

8.3.1 Connect Callback Path (–cb-connect)

The subpath for the connect callback is specified by –cb-connect PATH; the default is "connect"

8.3.2 Disconnect Callback Path (–cb-disconnect)

The subpath for the connect callback is specified by –cb-disconnect PATH; the default is "disconnect"

8.3.3 Create Channel Callback Path (–cb-create_channel)

The subpath for the create_channel callback is specified by –cb-create_channel PATH; the default is "create_channel"

8.3.4 Destroy Channel Callback Path (–cb-destroy_channel)

The subpath for the destroy_channel callback is specified by –cb-destroy_channel PATH; the default is "destroy_channel"

8.3.5 Subscribe Callback Path (–cb-subscribe)

The subpath for the subscribe callback is specified by –cb-subscribe PATH; the default is "subscribe"

8.3.6 Unsubscribe Callback Path (–cb-unsubscribe)

The subpath for the unsubscribe callback is specified by –cb-unsubscribe PATH; the default is “unsubscribe”

8.3.7 Publish Callback Path (–cb-publish)

The subpath for the publish callback is specified by –cb-publish PATH; the default is “publish”

8.3.8 Cookie Identifier (-c, –cookie-identifier)

Hookbox will include all user cookies in any user-triggered webhook callback. This option is purely an optimization that will cause hookbox to include only the cookie specified by -c COOKIE_NAME or –cookie-identifier COOKIE_NAME; the default is to include all cookies.

8.4 API Options

8.4.1 Web API Port (-w, –web-api-port)

Optionally bind web api listening socket to a different port, (default:none)

8.4.2 Web API Interface (-W, –web-api-interface)

Optionally bind web api listening socket to a different interface, (default: none)

8.4.3 API Secret (-r, –api-security-token)

The external api interfaces are disabled by default and will only be enabled if an API secret is specified by -r SECRET or –api-security-token SECRET. The value specified must appear in the form as the value for the key “secret” when using the Web/HTTP Hookbox API.

8.5 Admin Options

8.5.1 Admin Password (-a, –admin-password)

Hookbox includes an admin console which can be found at the /admin relative url. (e.g. http://localhost:8001/admin) This console is disabled by default unless an admin password is specified by -a PASSWORD or –admin-password PASSWORD

CHAPTER

NINE

DEPLOYMENT