rabbitChat Documentation

Release 1.1.0

Anirban Roy DAs

Sep 27, 2017

Contents

A Chat-Server/Chat-System based on AMQP protocol(RabbitMQ Message Broker) written in python using Tornado and RabbitMQ. **Home Page :** https://pypi.python.org/pypi/rabbitChat

CHAPTER 1

Details

Author Anirban Roy Das

Email anirban.nick@gmail.com

CHAPTER 2

Documentation:

Overview

rabbitChat is a very simple Chat Server which can be set up locally to chat in your LAN. It supports both **Public Chat** among all participants connected simultaneously at a particular time and also **Private Chat** betweent those individual participants.

It uses the AMQP protocol to implement the real time message passing system. **AMQP** is implemented in many languages and in many softwares, once of such is RabbitMQ , which is a message broker implementing the AMQP protocol.

The connection is created using the sockjs protocol. **SockJS** is implemented in many languages, primarily in Javascript to talk to the servers in real time, which tries to create a duplex bi-directional connection between the **Client(browser)** and the **Server**. Ther server should also implement the **sockjs** protocol. Thus using the sockjs-tornado library which exposes the **sockjs** protocol in Tornado server.

It first tries to create a Websocket connection, and if it fails then it fallbacks to other transport mechanisms, such as **Ajax**, **long polling**, etc. After the connection is established, the tornado server**(sockjs-tornado)** connects to **rabbitMQ** via AMQP protocol using the **AMQP Python Client Library**, Pika.

Thus the connection is *web-browser* to *tornado* to *rabbitMQ* and vice versa.

Features

Technical Specs

sockjs-client Advanced Websocket Javascript Clien

Tornado Async Python Web Library + Web Server

sockjs-tornado SockJS websocket server implementation for Tornado

AMQP Advance Message Queuing Protocol used in Message Oriented Middleware

pika AMQP Python Client Library

RabbitMQ A Message Broker implementing AMQP

pytest Python testing library and test runner with awesome test discobery

pytest-flask Pytest plugin for flask apps, to test fask apps using pytest library.

Uber's Test-Double Test Double library for python, a good alternative to the mock library

Jenkins (Optional) A Self-hosted CI server

Travis-CI (Optional) A hosted CI server free for open-source projecs

Docker A containerization tool for better devops

Feature Specs

- Public chat
- Shows who joined and who left
- Shows number of people online
- Shows who is typing and who is not
- Join/Leave chat room features
- Microservice
- Testing using Docker and Docker Compose
- CI servers like Jenkins, Travis-CI

Installation

There are two types of Installation. One using rabbitChat as a binary by installaing from pip and running the application in the local machine directly. Another method is running the application from Docker. Hence another set of installation steps for the Docker use case.

[Docker Method] Prerequisite (Optional)

To safegurad secret and confidential data leakage via your git commits to public github repo, check `git-secrets`.

This git secrets project helps in preventing secrete leakage by mistake.

[Docker Method] Dependencies

1. Docker
2. Make (Makefile)

See, there are so many technologies used mentioned in the tech specs and yet the dependencies are just two. This is the power of Docker.

[Docker Method] Install

- **Step 1 - Install Docker**

  Follow my another github project, where everything related to DevOps and scripts are mentioned along with setting up a development environemt to use Docker is mentioned.

  - Project: https://github.com/anirbanroydas/DevOps
  - Go to setup directory and follow the setup instructions for your own platform, linux/macos

- **Step 2 - Install Make**

```
# (Mac Os)
$ brew install automake

# (Ubuntu)
$ sudo apt-get update
$ sudo apt-get install make
```

- **Step 3 - Install Dependencies**

  Install the following dependencies on your local development machine which will be used in various scripts.

  1. openssl
  2. ssh-keygen
  3. openssh

[Standalone Binary Method] Prerequisites

1. python 2.7+
2. tornado
3. sockjs-tornado
4. sockjs-client
5. pika
6. rabbitMQ

[Standalone Binary Method] Install

```
$ pip install rabbitChat
```

If above dependencies do not get installed by the above command, then use the below steps to install them one by one.

Step 1 - Install pip

Follow the below methods for installing pip. One of them may help you to install pip in your system.

- **Method 1 -** https://pip.pypa.io/en/stable/installing/
- **Method 2 -** http://ask.xmodulo.com/install-pip-linux.html
- **Method 3 -** If you installed python on MAC OS X via `brew install python`, then **pip** is already installed along with python.

Step 2 - Install tornado

```
$ pip install tornado
```

Step 3 - Install sockjs-tornado

```
$ pip install sockjs-tornado
```

Step 4 - Install pika

```
$ pip install pika
```

Step 5 - Install RabbitMQ

- *For* `Mac` *Users*

  1. Brew Install RabbitMQ

```
$ brew install rabbitmq
```

  2. Configure RabbitMq, follow this link, this one and this.

- *For* `Ubuntu/Linux` *Users*

  1. Enable RabbitMQ application repository

```
$ echo "deb http://www.rabbitmq.com/debian/ testing main" >> /etc/apt/
↪sources.list
```

  2. Add the verification key for the package

```
$ wget -o http://www.rabbitmq.com/rabbitmq-signing-key-public.asc | sudo␣
↪apt-key add -
```

  3. Update the sources with our new addition from above

```
$ apt-get update
```

  4. And finally, download and install RabbitMQ

```
$ sudo apt-get install rabbitmq-server
```

  5. Configure RabbitMQ, follow this link, this one and this.

CI Setup

If you are using the project in a CI setup (like travis, jenkins), then, on every push to github, you can set up your travis build or jenkins pipeline. Travis will use the `.travis.yml` file and Jenknis will use the `Jenkinsfile` to do their jobs. Now, in case you are using Travis, then run the Travis specific setup commands and for Jenkins run the Jenkins specific setup commands first. You can also use both to compare between there performance.

The setup keys read the values from a `.env` file which has all the environment variables exported. But you will notice an example `env` file and not a `.env` file. Make sure to copy the `env` file to `.env` and **change/modify** the actual variables with your real values.

The `.env` files are not commited to git since they are mentioned in the `.gitignore` file to prevent any leakage of confidential data.

After you run the setup commands, you will be presented with a number of secure keys. Copy those to your config files before proceeding.

NOTE: This is a one time setup. **NOTE:** Check the setup scripts inside the `scripts/` directory to understand what are the environment variables whose encrypted keys are provided. **NOTE:** Don't forget to **Copy** the secure keys to your `.travis.yml` or `Jenkinsfile`

NOTE: If you don't want to do the copy of `env` to `.env` file and change the variable values in `.env` with your real values then you can just edit the `travis-setup.sh` or `jenknis-setup.sh` script and update the values their directly. The scripts are in the `scripts/` project level directory.

IMPORTANT: You have to run the `travis-setup.sh` script or the `jenkins-setup.sh` script in your local machine before deploying to remote server.

Travis Setup

These steps will encrypt your environment variables to secure your confidential data like api keys, docker based keys, deploy specific keys.

```
$ make travis-setup
```

Jenkins Setup

These steps will encrypt your environment variables to secure your confidential data like api keys, docker based keys, deploy specific keys.

```
$ make jenkins-setup
```

Usage

There are two types of Usage. One using rabbitChat as a binary by installaing from pip and running the application in the local machine directly. Another method is running the application from Docker. Hence another set of usage steps for the Docker use case.

[Docker Method]

After having installed the above dependencies, and ran the **Optional** (If not using any CI Server) or **Required** (If using any CI Server) **CI Setup** Step, then just run the following commands to use it:

You can run and test the app in your local development machine or you can run and test directly in a remote machine. You can also run and test in a production environment.

[Docker Method] Run

The below commands will start everythin in development environment. To start in a production environment, suffix `-prod` to every **make** command.

For example, if the normal command is `make start`, then for production environment, use `make start-prod`. Do this modification to each command you want to run in production environment.

Exceptions: You cannot use the above method for test commands, test commands are same for every environment. Also the `make system-prune` command is standalone with no production specific variation (Remains same in all environments).

- **Start Applcation**

```
$ make clean
$ make build
$ make start

# OR

$ docker-compose up -d
```

- **Stop Application**

```
$ make stop

# OR

$ docker-compose stop
```

- **Remove and Clean Application**

```
$ make clean

# OR

$ docker-compose rm --force -v
$ echo "y" | docker system prune
```

- **Clean System**

```
$ make system-prune

# OR

$ echo "y" | docker system prune
```

[Docker Method] Logging

- To check the whole application Logs

```
$ make check-logs

# OR

$ docker-compose logs --follow --tail=10
```

- To check just the python app's logs

```
$ make check-logs-app

# OR

$ docker-compose logs --follow --tail=10 identidock
```

[Standalone Binary Method] Run

After having installed rabbitChat via pip, just the run the following commands to use it:

- **RabbitMQ Server**

  1. *For* `Mac` *Users*

```
# start normally
$ rabbitmq-server

# If you want to run in background
$ rabbitmq-server --detached

# start using brew rervices (doesn't work with tmux)
$ brew services rabbitmq start
```

  2. *For* `Ubuntu/LInux` *Users*

```
# start normally
$ rabbitmq-server

# If you want to run in background
$ rabbitmq-server --detached

# To start using service
$ service rabbitmq-server start

# To stop using service
$ service rabbitmq-server stop

# To restart using service
$ service rabbitmq-server restart

# To check the status
$ service rabbitmq-server status
```

- **Start rabbitChat Server**

```
$ rabbitChat [options]
```

  - **Options**

    –port Port number where the chat server will start

  - **Example**

```
$ rabbitChat --port=9191
```

- **Stop rabbitChat Server**

  Click `Ctrl+C` to stop the server.

API

This contains all the modules and classes used to make the app. You can go through each of them for better understanding of the project.

Main View

This is the main view module which manages main tornado connections. This module provides request handlers for managing simple HTTP requests as well as Websocket requests.

Although the websocket requests are actually sockJs requests which follows the sockjs protcol, thus it provide interface to sockjs connection handlers behind the scene.

IndexHandler

class rabbitChat.apps.main.views.**IndexHandler**(*application*, *request*, ***kwargs*)

This handler is a basic regular HTTP handler to serve the chatroom page.

get()

This method is called when a client does a simple GET request, all other HTTP requests like POST, PUT, DELETE, etc are ignored.

Returns Returns the rendered main requested page, in this case its the chat page, index.html

SUPPORTED_METHODS = ('GET', 'HEAD', 'POST', 'DELETE', 'PATCH', 'PUT', 'OPTIONS')

_ARG_DEFAULT = <object object>

_INVALID_HEADER_CHAR_RE = <_sre.SRE_Pattern object>

__class__

alias of type

__delattr__

x.__delattr__('name') <==> del x.name

__dict__ = dict_proxy({'__module__': 'rabbitChat.apps.main.views', '__doc__': 'This handler is a basic regular HTTP

__format__()

default object formatter

__getattribute__

x.__getattribute__('name') <==> x.name

__hash__

__init__(*application*, *request*, ***kwargs*)

__module__ = 'rabbitChat.apps.main.views'

__new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()

helper for pickle

__reduce_ex__()

helper for pickle

__repr__

__setattr__

x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int

size of object in memory, in bytes

__str__

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

_break_cycles()

_clear_headers_for_304()

_convert_header_value(*value*)

_decode_xsrf_token(*cookie*)
Convert a cookie string into a the tuple form returned by _get_raw_xsrf_token.

_execute(**args*, ***kwargs*)
Executes this request with the given output transforms.

_get_argument(*name*, *default*, *source*, *strip=True*)

_get_arguments(*name*, *source*, *strip=True*)

_get_raw_xsrf_token()
Read or generate the xsrf token in its raw form.

The raw_xsrf_token is a tuple containing:

- version: the version of the cookie from which this token was read, or None if we generated a new token in this request.
- token: the raw token data; random (non-ascii) bytes.
- timestamp: the time this token was generated (will not be accurate for version 1 cookies)

_handle_request_exception(*e*)

_log()
Logs the current request.

Sort of deprecated since this functionality was moved to the Application, but left in place for the benefit of existing apps that have overridden this method.

_remove_control_chars_regex = <_sre.SRE_Pattern object>

_request_summary()

_stack_context_handle_exception(*type*, *value*, *traceback*)

_template_loader_lock = <thread.lock object>

_template_loaders = {}

_ui_method(*method*)

_ui_module(*name*, *module*)

add_header(*name*, *value*)
Adds the given response header and value.

Unlike *set_header*, *add_header* may be called multiple times to return multiple values for the same header.

check_etag_header()
Checks the `Etag` header against requests's `If-None-Match`.

Returns `True` if the request's Etag matches and a 304 should be returned. For example:

```
self.set_etag_header()
if self.check_etag_header():
    self.set_status(304)
    return
```

This method is called automatically when the request is finished, but may be called earlier for applications that override *compute_etag* and want to do an early check for `If-None-Match` before completing the request. The `Etag` header should be set (perhaps with *set_etag_header*) before calling this method.

check_xsrf_cookie()
Verifies that the `_xsrf` cookie matches the `_xsrf` argument.

To prevent cross-site request forgery, we set an `_xsrf` cookie and include the same value as a non-cookie field with all `POST` requests. If the two do not match, we reject the form submission as a potential forgery.

The `_xsrf` value may be set as either a form field named `_xsrf` or in a custom HTTP header named `X-XSRFToken` or `X-CSRFToken` (the latter is accepted for compatibility with Django).

See http://en.wikipedia.org/wiki/Cross-site_request_forgery

Prior to release 1.1.1, this check was ignored if the HTTP header `X-Requested-With: XMLHTTPRequest` was present. This exception has been shown to be insecure and has been removed. For more information please see http://www.djangoproject.com/weblog/2011/feb/08/security/ http://weblog.rubyonrails.org/2011/2/8/csrf-protection-bypass-in-ruby-on-rails

Changed in version 3.2.2: Added support for cookie version 2. Both versions 1 and 2 are supported.

clear()
Resets all headers and content for this response.

clear_all_cookies(*path='/'*, *domain=None*)
Deletes all the cookies the user sent with this request.

See *clear_cookie* for more information on the path and domain parameters.

Changed in version 3.2: Added the `path` and `domain` parameters.

clear_cookie(*name*, *path='/'*, *domain=None*)
Deletes the cookie with the given name.

Due to limitations of the cookie protocol, you must pass the same path and domain to clear a cookie as were used when that cookie was set (but there is no way to find out on the server side which values were used for a given cookie).

clear_header(*name*)
Clears an outgoing header, undoing a previous *set_header* call.

Note that this method does not apply to multi-valued headers set by *add_header*.

compute_etag()
Computes the etag header to be used for this request.

By default uses a hash of the content written so far.

May be overridden to provide custom etag implementations, or may return None to disable tornado's default etag support.

cookies
An alias for *self.request.cookies* <*.httputil.HTTPServerRequest.cookies*>.

create_signed_value(*name*, *value*, *version=None*)
Signs and timestamps a string so it cannot be forged.

Normally used via set_secure_cookie, but provided as a separate method for non-cookie uses. To decode a value not stored as a cookie use the optional value argument to get_secure_cookie.

Changed in version 3.2.1: Added the `version` argument. Introduced cookie version 2 and made it the default.

create_template_loader(*template_path*)
Returns a new template loader for the given path.

May be overridden by subclasses. By default returns a directory-based loader on the given path, using the `autoescape` and `template_whitespace` application settings. If a `template_loader` application setting is supplied, uses that instead.

current_user
The authenticated user for this request.

This is set in one of two ways:

- A subclass may override *get_current_user()*, which will be called automatically the first time `self.current_user` is accessed. *get_current_user()* will only be called once per request, and is cached for future access:

```
def get_current_user(self):
    user_cookie = self.get_secure_cookie("user")
    if user_cookie:
        return json.loads(user_cookie)
    return None
```

- It may be set as a normal variable, typically from an overridden *prepare()*:

```
@gen.coroutine
def prepare(self):
    user_id_cookie = self.get_secure_cookie("user_id")
    if user_id_cookie:
        self.current_user = yield load_user(user_id_cookie)
```

Note that *prepare()* may be a coroutine while *get_current_user()* may not, so the latter form is necessary if loading the user requires asynchronous operations.

The user object may be any type of the application's choosing.

data_received(*chunk*)
Implement this method to handle streamed request data.

Requires the *.stream_request_body* decorator.

decode_argument(*value*, *name=None*)
Decodes an argument from the request.

The argument has been percent-decoded and is now a byte string. By default, this method decodes the argument as utf-8 and returns a unicode string, but this may be overridden in subclasses.

This method is used as a filter for both *get_argument()* and for values extracted from the url and passed to *get()*/*post()*/etc.

The name of the argument is provided if known, but may be None (e.g. for unnamed groups in the url regex).

delete(**args*, ***kwargs*)

finish(*chunk=None*)
Finishes this response, ending the HTTP request.

flush(*include_footers=False*, *callback=None*)
Flushes the current output buffer to the network.

The `callback` argument, if given, can be used for flow control: it will be run when all flushed data has been written to the socket. Note that only one flush callback can be outstanding at a time; if another flush occurs before the previous flush's callback has been run, the previous callback will be discarded.

Changed in version 4.0: Now returns a *.Future* if no callback is given.

get_argument(*name*, *default=<object object>*, *strip=True*)
Returns the value of the argument with the given name.

If default is not provided, the argument is considered to be required, and we raise a *MissingArgumentError* if it is missing.

If the argument appears in the url more than once, we return the last value.

The returned value is always unicode.

get_arguments(*name*, *strip=True*)
Returns a list of the arguments with the given name.

If the argument is not present, returns an empty list.

The returned values are always unicode.

get_body_argument(*name*, *default=<object object>*, *strip=True*)
Returns the value of the argument with the given name from the request body.

If default is not provided, the argument is considered to be required, and we raise a *MissingArgumentError* if it is missing.

If the argument appears in the url more than once, we return the last value.

The returned value is always unicode.

New in version 3.2.

get_body_arguments(*name*, *strip=True*)
Returns a list of the body arguments with the given name.

If the argument is not present, returns an empty list.

The returned values are always unicode.

New in version 3.2.

get_browser_locale(*default='en_US'*)
Determines the user's locale from `Accept-Language` header.

See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.4

get_cookie(*name*, *default=None*)
Gets the value of the cookie with the given name, else default.

get_current_user()
Override to determine the current user from, e.g., a cookie.

This method may not be a coroutine.

get_login_url()
Override to customize the login URL based on the request.

By default, we use the `login_url` application setting.

get_query_argument(*name*, *default=<object object>*, *strip=True*)
Returns the value of the argument with the given name from the request query string.

If default is not provided, the argument is considered to be required, and we raise a *MissingArgumentError* if it is missing.

If the argument appears in the url more than once, we return the last value.

The returned value is always unicode.

New in version 3.2.

get_query_arguments(*name*, *strip=True*)
Returns a list of the query arguments with the given name.

If the argument is not present, returns an empty list.

The returned values are always unicode.

New in version 3.2.

get_secure_cookie(*name*, *value=None*, *max_age_days=31*, *min_version=None*)
Returns the given signed cookie if it validates, or None.

The decoded cookie value is returned as a byte string (unlike *get_cookie*).

Changed in version 3.2.1: Added the `min_version` argument. Introduced cookie version 2; both versions 1 and 2 are accepted by default.

get_secure_cookie_key_version(*name*, *value=None*)
Returns the signing key version of the secure cookie.

The version is returned as int.

get_status()
Returns the status code for our response.

get_template_namespace()
Returns a dictionary to be used as the default template namespace.

May be overridden by subclasses to add or modify values.

The results of this method will be combined with additional defaults in the *tornado.template* module and keyword arguments to *render* or *render_string*.

get_template_path()
Override to customize template path for each handler.

By default, we use the `template_path` application setting. Return None to load templates relative to the calling file.

get_user_locale()
Override to determine the locale from the authenticated user.

If None is returned, we fall back to *get_browser_locale()*.

This method should return a *tornado.locale.Locale* object, most likely obtained via a call like `tornado.locale.get("en")`

head(**args*, ***kwargs*)

initialize()
Hook for subclass initialization. Called for each request.

A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

Example:

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...

app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

locale
The locale for the current session.

Determined by either *get_user_locale*, which you can override to set the locale based on, e.g., a user preference stored in a database, or *get_browser_locale*, which uses the `Accept-Language` header.

log_exception(*typ*, *value*, *tb*)
Override to customize logging of uncaught exceptions.

By default logs instances of *HTTPError* as warnings without stack traces (on the `tornado.general` logger), and all other exceptions as errors with stack traces (on the `tornado.application` logger).

New in version 3.1.

on_connection_close()
Called in async handlers if the client closed the connection.

Override this to clean up resources associated with long-lived connections. Note that this method is called only if the connection was closed during asynchronous processing; if you need to do cleanup after every request override *on_finish* instead.

Proxies may keep a connection open for a time (perhaps indefinitely) after the client has gone away, so this method may not be called promptly after the end user closes their connection.

on_finish()
Called after the end of a request.

Override this method to perform cleanup, logging, etc. This method is a counterpart to *prepare*. `on_finish` may not produce any output, as it is called after the response has been sent to the client.

options(**args*, ***kwargs*)

patch(**args*, ***kwargs*)

post(**args*, ***kwargs*)

prepare()
Called at the beginning of a request before *get*/*post*/etc.

Override this method to perform common initialization regardless of the request method.

Asynchronous support: Decorate this method with *.gen.coroutine* or *.return_future* to make it asynchronous (the *asynchronous* decorator cannot be used on *prepare*). If this method returns a *.Future* execution will not proceed until the *.Future* is done.

New in version 3.1: Asynchronous support.

put(**args*, ***kwargs*)

redirect(*url*, *permanent=False*, *status=None*)
Sends a redirect to the given (optionally relative) URL.

If the `status` argument is specified, that value is used as the HTTP status code; otherwise either 301 (permanent) or 302 (temporary) is chosen based on the `permanent` argument. The default is 302 (temporary).

render(*template_name*, ***kwargs*)
Renders the template with the given arguments as the response.

render_embed_css(*css_embed*)
Default method used to render the final embedded css for the rendered webpage.

Override this method in a sub-classed controller to change the output.

render_embed_js(*js_embed*)
Default method used to render the final embedded js for the rendered webpage.

Override this method in a sub-classed controller to change the output.

render_linked_css(*css_files*)
Default method used to render the final css links for the rendered webpage.

Override this method in a sub-classed controller to change the output.

render_linked_js(*js_files*)
Default method used to render the final js links for the rendered webpage.

Override this method in a sub-classed controller to change the output.

render_string(*template_name*, ***kwargs*)
Generate the given template with the given arguments.

We return the generated byte string (in utf8). To generate and write a template as a response, use render() above.

require_setting(*name*, *feature='this feature'*)
Raises an exception if the given app setting is not defined.

reverse_url(*name*, **args*)
Alias for *Application.reverse_url*.

send_error(*status_code=500*, ***kwargs*)
Sends the given HTTP error code to the browser.

If *flush()* has already been called, it is not possible to send an error, so this method will simply terminate the response. If output has been written but not yet flushed, it will be discarded and replaced with the error page.

Override *write_error()* to customize the error page that is returned. Additional keyword arguments are passed through to *write_error*.

set_cookie(*name*, *value*, *domain=None*, *expires=None*, *path='/'*, *expires_days=None*, ***kwargs*)
Sets the given cookie name/value with the given options.

Additional keyword arguments are set on the Cookie.Morsel directly. See https://docs.python.org/2/library/cookie.html#Cookie.Morsel for available attributes.

set_default_headers()
Override this to set HTTP headers at the beginning of the request.

For example, this is the place to set a custom `Server` header. Note that setting such headers in the normal flow of request processing may not do what you want, since headers may be reset during error handling.

set_etag_header()
Sets the response's Etag header using `self.compute_etag()`.

Note: no header will be set if `compute_etag()` returns `None`.

This method is called automatically when the request is finished.

set_header(*name*, *value*)
Sets the given response header name and value.

If a datetime is given, we automatically format it according to the HTTP specification. If the value is not a string, we convert it to a string. All header values are then encoded as UTF-8.

set_secure_cookie(*name*, *value*, *expires_days=30*, *version=None*, ***kwargs*)
Signs and timestamps a cookie so it cannot be forged.

You must specify the `cookie_secret` setting in your Application to use this method. It should be a long, random sequence of bytes to be used as the HMAC secret for the signature.

To read a cookie set with this method, use *get_secure_cookie()*.

Note that the `expires_days` parameter sets the lifetime of the cookie in the browser, but is independent of the `max_age_days` parameter to *get_secure_cookie*.

Secure cookies may contain arbitrary byte values, not just unicode strings (unlike regular cookies)

Changed in version 3.2.1: Added the `version` argument. Introduced cookie version 2 and made it the default.

set_status(*status_code*, *reason=None*)
Sets the status code for our response.

Parameters

- **status_code** (*int*) – Response status code. If `reason` is `None`, it must be present in *httplib.responses* <*http.client.responses*>.
- **reason** (*string*) – Human-readable reason phrase describing the status code. If `None`, it will be filled in from *httplib.responses* <*http.client.responses*>.

settings
An alias for *self.application.settings* <*Application.settings*>.

static_url(*path*, *include_host=None*, ***kwargs*)
Returns a static URL for the given relative static file path.

This method requires you set the `static_path` setting in your application (which specifies the root directory of your static files).

This method returns a versioned url (by default appending `?v=<signature>`), which allows the static files to be cached indefinitely. This can be disabled by passing `include_version=False` (in the default implementation; other static file implementations are not required to support this, but they may support other options).

By default this method returns URLs relative to the current host, but if `include_host` is true the URL returned will be absolute. If this handler has an `include_host` attribute, that value will be used as the default for all *static_url* calls that do not pass `include_host` as a keyword argument.

write(*chunk*)
Writes the given chunk to the output buffer.

To write the output to the network, use the flush() method below.

If the given chunk is a dictionary, we write it as JSON and set the Content-Type of the response to be `application/json`. (if you want to send JSON as a different `Content-Type`, call set_header *after* calling write()).

Note that lists are not converted to JSON because of a potential cross-site security vulnerability. All JSON output should be wrapped in a dictionary. More details at http://haacked.com/archive/2009/06/25/json-hijacking.aspx/ and https://github.com/facebook/tornado/issues/1009

write_error(*status_code*, ***kwargs*)
Override to implement custom error pages.

`write_error` may call *write*, *render*, *set_header*, etc to produce output as usual.

If this error was caused by an uncaught exception (including HTTPError), an `exc_info` triple will be available as `kwargs["exc_info"]`. Note that this exception may not be the "current" exception for purposes of methods like `sys.exc_info()` or `traceback.format_exc`.

xsrf_form_html()
An HTML `<input/>` element to be included with all POST forms.

It defines the `_xsrf` input value, which we check on all POST requests to prevent cross-site request forgery. If you have set the `xsrf_cookies` application setting, you must include this HTML within all of your HTML forms.

In a template, this method should be called with `{% module xsrf_form_html() %}`

See *check_xsrf_cookie()* above for more information.

xsrf_token
The XSRF-prevention token for the current user/session.

To prevent cross-site request forgery, we set an '_xsrf' cookie and include the same '_xsrf' value as an argument with all POST requests. If the two do not match, we reject the form submission as a potential forgery.

See http://en.wikipedia.org/wiki/Cross-site_request_forgery

Changed in version 3.2.2: The xsrf token will now be have a random mask applied in every request, which makes it safe to include the token in pages that are compressed. See http://breachattack.com for more information on the issue fixed by this change. Old (version 1) cookies will be converted to version 2 when this method is called unless the `xsrf_cookie_version` *Application* setting is set to 1.

Changed in version 4.3: The `xsrf_cookie_kwargs` *Application* setting may be used to supply additional cookie options (which will be passed directly to *set_cookie*). For example, `xsrf_cookie_kwargs=dict(httponly=True, secure=True)` will set the `secure` and `httponly` flags on the `_xsrf` cookie.

ChatWebsocketHandler

class `rabbitChat.apps.main.views.`**ChatWebsocketHandler**(*session*)
Websocket Handler implementing the sockjs Connection Class which will handle the websocket/sockjs connections.

on_open(*info*)
This method is called when a websocket/sockjs connection is opened for the first time.

:param self The object :param info The information

Returns It returns the websocket object

on_message(*message*)
This method is called when a message is received via the websocket/sockjs connection created initially.

:param self The object :param message The message received via the connection.

Returns Returns the published message back to all other subscribers.

on_close()
This method is called when a websocket/sockjs connection is closed.

:param self The object

Returns Doesn't return anything, except a confirmation of closed connection back to web app.

genid()

__class__
alias of type

__delattr__
x.__delattr__('name') <==> del x.name

__dict__ = dict_proxy({'__module__': 'rabbitChat.apps.main.views', 'on_message': <function on_message>, 'genid': <

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__

__init__(*session*)
Connection constructor.

session Associated session

__module__ = 'rabbitChat.apps.main.views'

__new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

broadcast(*clients*, *message*)
Broadcast message to the one or more clients. Use this method if you want to send same message to lots of clients, as it contains several optimizations and will work fast than just having loop in your code.

clients Clients iterable

message Message to send.

close()

is_closed
Check if connection was closed

send(*message*, *binary=False*)
Send message to the client.

message Message to send.

PubSub Module

The pubsub module provides interface for the rabbitmq client.

It provides classes to create pika clients to connect to rabbitmq broker server, interact with and publish/subscribe to rabbitmq via creating channels, methods to publish, subscribe/consume, stop consuming, start publishing, start connection, stop connection, create channel, close channel, acknowledge delivery by publisher, acknowledge receiving of messages by consumers, send basic ack, basic cancel requests and also add callbacks for various other events.

RabbitMqClient

class rabbitChat.apps.rabbitmq.pubsub.**RabbitMqClient**(*credentials=None*, *params=None*, *queue=None*)
This is a RabbitMQ Client using the TornadoConnection Adapter that will handle unexpected interactions with RabbitMQ such as channel and connection closures.

If RabbitMQ closes the connection, it will reopen it. You should look at the output, as there are limited reasons why the connection may be closed, which usually are tied to permission related issues or socket timeouts.

If the channel is closed, it will indicate a problem with one of the commands that were issued and that should surface in the output as well.

It alos uses delivery confirmations and illustrates one way to keep track of messages that have been sent and if they've been confirmed by RabbitMQ.

__init__(*credentials=None*, *params=None*, *queue=None*)
Create a new instance of the consumer class, passing in the AMQP URL used to connect to RabbitMQ.

Parameters

- **credentials** (*pika.credentials.PlainCredentials*) – credentials to connect to rabbitmq broker server
- **params** (*pika.connection.ConnectionParameters*) – connection parameters used to connect with rabbitmq broker server
- **queue** (*string - random long base64 url safe encoded string*) – queue to be created after a channel is established which will be bound to an exchange

connect()
This method connects to RabbitMQ via the Torando Connectoin Adapter, returning the connection handle.

When the connection is established, the on_connection_open method will be invoked by pika.

Returns Returns a pika connection object which is a tornado connection object to rabbitmq server

Return type pika.adapters.TornadoConnection

on_connection_opened(*connection*)
This method is called by pika once the connection to RabbitMQ has been established. It passes the handle to the connection object in case we need it, but in this case, we'll just mark it unused.

Parameters **connection** (*pika.adapters.TornadoConnection*) – connection object created

add_on_connection_close_callback()
This method adds an on close callback that will be invoked by pika when RabbitMQ closes the connection to the publisher unexpectedly.

on_connection_closed(*connection*, *reply_code*, *reply_text*)
This method is invoked by pika when the connection to RabbitMQ is closed unexpectedly. Since it is unexpected, we will reconnect to RabbitMQ if it disconnects.

Parameters

- **connection** (*pika.connection.Connection*) – The closed connection obj
- **reply_code** (*int*) – The server provided reply_code if given
- **reply_text** (*str*) – The server provided reply_text if given

reconnect()
Will be invoked by the IOLoop timer if the connection is closed. See the on_connection_closed method.

close_connection()
This method closes the connection to RabbitMQ.

open_channel()
Open a new channel with RabbitMQ by issuing the Channel.Open RPC command. When RabbitMQ responds that the channel is open, the on_channel_open callback will be invoked by pika.

on_channel_open(*channel*)
This method is invoked by pika when the channel has been opened. The channel object is passed in so we can make use of it.

Since the channel is now open, we'll declare the exchange to use.

Parameters **channel** (*pika.channel.Channel*) – The channel object

close_channel()
Call to close the channel with RabbitMQ cleanly by issuing the Channel.Close RPC command.

add_on_channel_close_callback()
This method tells pika to call the on_channel_closed method if RabbitMQ unexpectedly closes the channel.

on_channel_closed(*channel*, *reply_code*, *reply_text*)
Invoked by pika when RabbitMQ unexpectedly closes the channel. Channels are usually closed if you attempt to do something that violates the protocol, such as re-declare an exchange or queue with different parameters. In this case, we'll close the connection to shutdown the object.

Parameters

- **channel** (*pika.channel.Channel*) – The closed channel
- **reply_code** (*int*) – The numeric reason the channel was closed
- **reply_text** (*str*) – The text reason the channel was closed

setup_exchange()
Setup the exchange on RabbitMQ by invoking the Exchange.Declare RPC command. When it is complete, the on_exchange_declareok method will be invoked by pika.

on_exchange_declareok(*frame*)
Invoked by pika when RabbitMQ has finished the Exchange.Declare RPC command.

Parameters **frame** (*pika.Frame.Method*) – Exchange.DeclareOk response frame

setup_queue()
Setup the queue on RabbitMQ by invoking the Queue.Declare RPC command. When it is complete, the on_queue_declareok method will be invoked by pika.

on_queue_declareok(*method_frame*)
Method invoked by pika when the Queue.Declare RPC call made in setup_queue has completed. mand is complete, the on_bindok method will be invoked by pika.

Parameters **method_frame** (*pika.frame.Method*) – The Queue.DeclareOk frame

bind_queue(*binding_key*)
In this method we will bind the queue and exchange together with the routing key by issuing the Queue.Bind RPC command.

Parameters **binding_key** (*string*) – The routing_key argument

on_bindok(*unused_frame*)
Invoked by pika when the Queue.Bind method has completed. At this point we will start consuming messages by calling start_consuming which will invoke the needed RPC commands to start the process. It will also set channel for publishing.

Parameters **unused_frame** (*pika.frame.Method*) – The Queue.BindOk response frame

setup_publishing()
In this method we will setup the channel for publishing by making it available for delivery confirmations and publisher confirmations.

enable_delivery_confirmations()
Send the Confirm.Select RPC method to RabbitMQ to enable delivery confirmations on the channel. The only way to turn this off is to close the channel and create a new one.

When the message is confirmed from RabbitMQ, the on_delivery_confirmation method will be invoked passing in a Basic.Ack or Basic.Nack method from RabbitMQ that will indicate which messages it is confirming or rejecting.

on_delivery_confirmation(*method_frame*)
Invoked by pika when RabbitMQ responds to a Basic.Publish RPC command, passing in either a Basic.Ack or Basic.Nack frame with the delivery tag of the message that was published. The delivery tag is an integer counter indicating the message number that was sent on the channel via Basic.Publish. Here we're just doing house keeping to keep track of stats and remove message numbers that we expect a delivery confirmation of from the list used to keep track of messages that are pending confirmation.

Parameters **method_frame** (*pika.frame.Method*) – Basic.Ack or Basic.Nack frame

publish(*msg*, *routing_key*)
If the class is not stopping, publish a message to RabbitMQ, appending a list of deliveries with the message number that was sent. This list will be used to check for delivery confirmations in the on_delivery_confirmations method.

Once the message has been sent, schedule another message to be sent. The main reason I put scheduling in was just so you can get a good idea of how the process is flowing by slowing down and speeding up the delivery intervals by changing the PUBLISH_INTERVAL constant in the class.

Parameters

- **msg** – Message to be published to Channel

- **routing_key** (*string*) – Routing Key to direct message via the Exchange

Tyep msg string

start_consuming()
This method sets up the consumer by first calling add_on_cancel_callback so that the object is notified if RabbitMQ cancels the consumer. It then issues the Basic.Consume RPC command which returns the consumer tag that is used to uniquely identify the consumer with RabbitMQ. We keep the value to use it when we want to cancel consuming. The on_message method is passed in as a callback pika will invoke when a message is fully received.

add_on_cancel_callback()
Add a callback that will be invoked if RabbitMQ cancels the consumer for some reason. If RabbitMQ does cancel the consumer, on_consumer_cancelled will be invoked by pika.

on_consumer_cancelled(*method_frame*)
Invoked by pika when RabbitMQ sends a Basic.Cancel for a consumer receiving messages.

Parameters **method_frame** (*pika.frame.Method*) – The Basic.Cancel frame

on_message(*unused_channel*, *basic_deliver*, *properties*, *body*)
Invoked by pika when a message is delivered from RabbitMQ. The channel is passed for your convenience. The basic_deliver object that is passed in carries the exchange, routing key, delivery tag and a redelivered flag for the message. The properties passed in is an instance of BasicProperties with the message properties and the body is the message that was sent.

Parameters

- **unused_channel** (*pika.channel.Channel*) – The channel object
- **basic_deliver** (*pika.Spec.Basic.Deliver*) – The basic delivery object passed
- **properties** (*pika.Spec.BasicProperties*) – The basic properties used to publish the message
- **body** (*str|unicode*) – The message body

acknowledge_message(*delivery_tag*)
Acknowledge the message delivery from RabbitMQ by sending a Basic.Ack RPC method for the delivery tag.

Parameters **delivery_tag** (*int*) – The delivery tag from the Basic.Deliver frame

stop_consuming()
Tell RabbitMQ that you would like to stop consuming by sending the Basic.Cancel RPC command.

on_cancelok(*unused_frame*)
This method is invoked by pika when RabbitMQ acknowledges the cancellation of a consumer. At this point we will close the channel. This will invoke the on_channel_closed method once the channel has been closed, which will in-turn close the connection.

Parameters **unused_frame** (*pika.frame.Method*) – The Basic.CancelOk frame

start()
Run the example consumer by connecting to RabbitMQ and then starting the IOLoop to block and allow the SelectConnection to operate.

stop()
Cleanly shutdown the connection to RabbitMQ by stopping the consumer with RabbitMQ. When RabbitMQ confirms the cancellation, on_cancelok will be invoked by pika, which will then closing the channel and connection. The IOLoop is started again because this method is invoked when CTRL-C is pressed raising a KeyboardInterrupt exception. This exception stops the IOLoop which needs to be running for

pika to communicate with RabbitMQ. All of the commands issued prior to starting the IOLoop will be buffered but not processed.

status()
Gives the status of the RabbitMQClient Connection.

Returns Returns the current status of the connection

Return type self._status

__class__
alias of `type`

__delattr__
x.__delattr__('name') <==> del x.name

__dict__ = dict_proxy({'setup_queue': <function setup_queue>, '__module__': 'rabbitChat.apps.rabbitmq.pubsub', 'c

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__

__module__ = 'rabbitChat.apps.rabbitmq.pubsub'

__new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

Testing

NOTE: Testing is only done using the Docker Method. anyway, it should not matter whether you run your application using the Docker Method or the Standalone Method. Testing is independent of it.

Now, testing is the main deal of the project. You can test in many ways, namely, using `make` commands as mentioned in the below commands, which automates everything and you don't have to know anything else, like what test library

or framework is being used, how the tests are happening, either directly or via `docker` containers, or may be different virtual environments using `tox`. Nothing is required to be known.

On the other hand if you want fine control over the tests, then you can run them directly, either by using `pytest` commands, or via `tox` commands to run them in different python environments or by using `docker-compose` commands to run differetn tests.

But running the make commands is lawasy the go to strategy and reccomended approach for this project.

NOTE: Tox can be used directly, where `docker` containers will not be used. Although we can try to run `tox` inside our test contianers that we are using for running the tests using the `make` commands, but then we would have to change the `Dockerfile` and install all the `python` dependencies like `python2.7`, `python3.x` and then run `tox` commands from inside the `docker` containers which then run the `pytest` commands which we run now to perform our tests inside the current test containers.

CAVEAT: The only caveat of using the make commands directly and not using `tox` is we are only testing the project in a single `python` environment, nameley `python 3.6`.

- To Test everything

```
$ make test
```

  Any Other method without using make will involve writing a lot of commands. So use the make command preferrably

- To perform Unit Tests

```
$ make test-unit
```

- To perform Component Tests

```
$ make test-component
```

- To perform Contract Tests

```
$ make test-contract
```

- To perform Integration Tests

```
$ make test-integration
```

- To perform End To End (e2e) or System or UI Acceptance or Functional Tests

```
$ make test-e2e

# OR

$ make test-system

# OR

$ make test-ui-acceptance

# OR

$ make test-functional
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

Symbols

A

B

C

D

E

F

G

H

I

W

X