Pyjamas and Flask Examples

Release

July 18, 2015

Contents

1 Introduction **3**

2 Cross-Origin Resource Sharing Between a Webserver and Flask's Development Server **5**

3 Flask + Pyjamas + Celery **7**

3.1 Introduction 7

3.2 Dependencies 7

3.3 More Fun with Asynchronicity 8

Note: This example assumes that you already have Flask installed and working on your system (pip install flask).

Introduction

The goal is to have Flask and Pyjamas working together and make it somewhat easier to develop interactive applications that have a heavy computational load. The Flask application runs on the webserver, the Pyjamas application can run in a web browser (pyjs) or as a desktop application (pyjd), and the computational work can be done anywhere you can run a daemon process (or use screen) that can connect to the broker (managing process).

This document assumes that you know enough (or how to find it) with respect to configuring the web server of your choice to host the pyjs application and flask server code. Also, it expects that you understand the asynchronous nature of Pyjamas programs, and are familiar with the basics of building Flask applications. Before getting started you may wish to first understand some of the components separately by looking at the examples/jsonrpc directory, Flask's documentation, etc.

CHAPTER 2

Cross-Origin Resource Sharing Between a Webserver and Flask's Development Server

When building/debugging a Flask application, it is advantageous to use the built in webserver because debugging a Flask application running on a production webserver is very painful. In this document we will assume it will be run on the default location localhost:5000. The flask webserver's debugging mode + tracebacks are very helpful, and stdout for the application is printed directly in the server's output. Finally, it gives us an excuse to test Cross-Origin Resource Sharing (CORS) – you may find HTTP access control more suitable for reading than the W3C working draft.

The key ideas are:

- We will be extending the pyjamas examples/jsonrpc code
- We continue to use the JSONProxy/JSONService proxies used by the examples/jsonrpc code
- At a lower level, we are basically issuing a javascript XMLHttpRequest (XHR) for each JSONRPC call.
  - Most modern browsers will preflight JSON-RPC requests (see CORS and HTTP access control).
  - From Pyjamas we must handle the results via callbacks due to the asynchronous nature of XHR, but nothing else since we are at the mercy of the users' browsers' implementation of XHR.
  - From the server's (Flask application's) point of view, we must correctly handle the OPTIONS method or the browser will complain about Cross-Domain/Cross-Origin XHR and ignore any responses.

You may find the modified examples/jsonrpc/JSONRPCExample.py as Flask_JSONRPC_CORS.py. Please compile this to javascript:

```
./build.sh
```

Copy the output directory to a directory hosted by your web server. As an example, on Mac OSX, one may copy the output directory to a subdirectory of /Library/Webserver/Documents to have Apache2 serve the pyjamas app:

```
cp -r output /Library/Webserver/Documents/pyjamas_examples/Flask_CORS
```

To verify that this works you would then browse to localhost/pyjamas_examples/Flask_CORS/Flask_JSONRPC_CORS.html and click on the PHP buttons (provided you have Apache allowed to run PHP files).

The Flask server can be started by going to the public/services/flask_example/directory and executing:

```
python run_flask_server.py
```

This will start the flask server on localhost:5000. You may now send JSON-RPC echo commands to the Flask application via the pyjamas application running in your browser. If you look carefully at the Flask server's output, you will notice that the first time you send a JSON-RPC command to the server it should receive and respond to an OPTIONS method.

If you wish to understand/extend the example, please look at:

- flaskcors/Flask_JSONRPC_CORS.py for the pyjamas code
- The directory flaskcors/public/services/flask_example for:
  - views.py: The Flask view function to handle OPTIONS and POST
  - requests.py: Subclassing the default Flask Request to support parsing the mimetype ‘application/json-rpc’
  - FlaskEchoApp.py: Setting/creating the Flask application
  - run_flask_server.py: Running the Flask server

CHAPTER 3

Flask + Pyjamas + Celery

Note: This example assumes that you already have Flask installed and working on your system (pip install flask), **and** you are willing to install **and** configure RabbitMQ and Celery.

3.1 Introduction

You may wish to have your webserver separate from the computational portion of your JSON-RPC or at the least some way to monitor/ration the resources given to the JSON-RPC over the rest of the Flask application. One method to do so is to use a distributed task queue such as Celery. There are many reasons for choosing one specific implementation or tool over another, and here the focus is on a specific way of using Celery with Flask and not whether another tool would be better suited or ...

3.2 Dependencies

Celery requires a tool for message passing (pub/sub, whatever you want to call it). In this example we will be using RabbitMQ. Therefore, we require installation of the following (and all dependencies pip pulls in).

3.2.1 RabbitMQ

First, install RabbitMQ and make sure it is running via either the HTTP plugin and/or rabbitmqctl. Please refer to the RabbitMQ Documentation for installing and configuring RabbitMQ.

Note: rabbitmqctl calls rabbitmq-env and rabbitmq-env puts together your node name based on $HOSTNAME. Therefore, if your $HOSTNAME is not set correctly (thanks Mac OSX), rabbitmqctl will fail with "nodedown blah blah" even when the web plugin works fine.

Other than the above note, I had no issues getting RabbitMQ up and running. Oh, wait ... unless you do the typical sysadmin steps of creating a _rabbitmq user and group, giving _rabbitmq an executable shell, and giving _rabbitmq a usable home directory, and creating the necessary log directories your mileage could vary greatly.

3.2.2 Celery

Celery has excellent Python support and has a script wrapper around Flask applications. You can install Flask-Celery and its dependencies via:

```
pip install flask-celery
```

I could use Celery out of the box. Before trying Celery + Flask, I recommend first looking at the tutorial Celery Tutorial so you have some understanding of how Celery works.

3.3 More Fun with Asynchronicity

By using Pyjamas, Flask, and Celery we can have multiple levels of asynchonicity. There is an included example that shows how one can use JSON-RPC from a pyjamas application to initiate an asynchonous job on a Flask server. Since we are using RabbitMQ, we can off-load the resource intensive RPC requests to machines other than the webserver. The primary point of this example is to be a proof-of-concept, and it is **not** intended to illustrate best practices for application scaling.

At this point, I would become familiar with the examples/timerdemo as the Pyjamas app uses a repeating timer is used to query the Flask server about the status of the echo.

As a small bonus, this example shows how to use the current Flask implementation for generating views from classes.

3.3.1 Celery Configuration

This took me some time to figure out as I was tired, and it wasn't blatantly obvious: "place the CELERY config values inside the Flask's application config (I chose to do this by placing the config values inside the celeryconfig.py file and configuring the app from that file)."

3.3.2 Building and Installing

You may find the pyjamas source as flaskcelery/FLASKCELERYExample.py. Please compile this to javascript:

```
cd flaskcelery
./build.sh
```

Copy the output directory to a directory hosted by your web server. As an example, on Mac OSX, one may copy the output directory to a subdirectory of /Library/Webserver/Documents to have Apache2 serve the pyjamas app:

```
cp -r output /Library/Webserver/Documents/pyjamas_examples/Flask_Celery
```

To verify that this works you would then browse to localhost/pyjamas_examples/Flask_CORS/FLASKCELERYExample.html and click on the PHP buttons (provided you have Apache allowed to run PHP files).

3.3.3 Running a Celery Worker and Flask

Use screen, or open two tabs, or open two windows, or do whatever you want ...

Edit the flask_source/celeryconfig.py file and change the values for *BROKER_USER* and *BROKER_PASSWORD* to a valid RabbitMQ user and corresponding password (by default you can use "guest" and "guest" if you have not disabled the guest account).

The Flask server can be started by going to the flask_source directory and executing:

```
python manage_celery.py runserver
```

A Celeryd (Celery Worker) instance can be started from the flask_source directory and executing:

```
python manage_celery.py celeryd --loglevel=DEBUG
```

Notice that the request takes some time to be handled since I set a delay of 5 seconds for the processing of the echo. Thus, you may browse to the pyjamas app and notice that after you click on the flask celery buttion you may see the number of times the pyjamas app had to query the Flask server before the celery job was finished. Upon finishing, the view is updated to denote the echo result was received, the Flask server output makes note of the HTTP OPTIONS and POSTS, and the celeryd worker prints debug info about the job.

If you wish to understand/extend the example, please look at:

- flaskcelery/FLASKCELERYExample.py for the pyjamas code
- The directory flaskcelery/flask_source for:
  - views.py: The Flask view function to handle OPTIONS and POST
  - method_views.py: The Flask view classes for JSON-RPC echo
  - requests.py: Subclassing the default Flask Request to support parsing the mimetype 'application/json-rpc'
  - FlaskEchoApp.py: Setting/creating the Flask application
  - celery_views.py: The Flask view class for Celery JSON-RPC echo
  - celery_tasks.py: The Celery tasks for workers to perform
  - celeryconfig.py: Flask and Celery config variables
  - manage_celery.py: Running the Flask server and Celery worker