django-websocket-redis Documentation

Release 0.5.0

Jacob Rief

Jun 07, 2017

Contents

1 Project's home **3**

2 Contents **5**

2.1 Introduction 5
2.2 Installation and Configuration 7
2.3 Running WebSocket for Redis 10
2.4 Using Websockets for Redis 15
2.5 Sending and receiving heartbeat messages 20
2.6 Application Programming Interface 21
2.7 Testing Websockets for Redis 23
2.8 Debugging 24
2.9 Release History 26
2.10 Credits to Others 29

3 Indices and tables **31**

This module implements websockets on top of Django without requiring any additional framework. For messaging it uses the Redis datastore. In a production environment, it is intended to work under uWSGI and behind NGiNX. In a development environment, it can be used with `manage runserver`.

CHAPTER 1

Project's home

Check for the latest release of this project on Github.

Please report bugs or ask questions using the Issue Tracker.

CHAPTER 2

Contents

Introduction

Application servers such as Django and Ruby-on-Rails have been developed without intention to create long living connections. Therefore these frameworks are not a good fit for web applications, which shall react on asynchronous events initiated by the server. One feasible solution for clients wishing to be notified for events is to continuously poll the server using an XMLHttpRequest (Ajax). This however produces a lot of traffic, and depending on the granularity of the polling interval, it is not a viable solution for real time events such as chat applications or browser based multiplayer games.

Web application written in Python usually use WSGI as the communication layer between the webserver and themselves. WSGI is a stateless protocol which defines how to handle requests and making responses in a simple way abstracted from the HTTP protocol, but by design it does not support non-blocking requests.

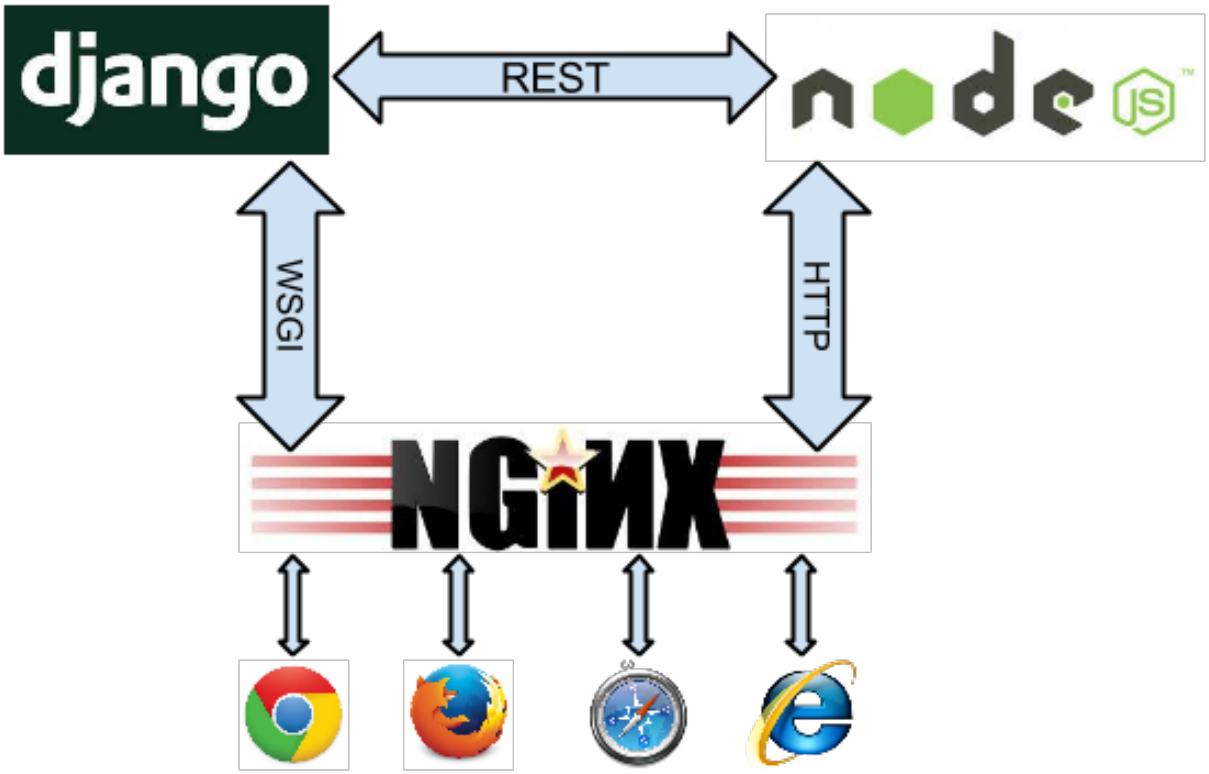
The WSGI protocol can not support websockets

In Django, the web server accepts an incoming request, sets up a WSGI dictionary which then is passed to the application server. There the HTTP headers and the payload is created and immediately afterwards the request is finished and flushed to the client. This processing typically requires a few dozen milliseconds. The throughput, such a server can handle, is the average response time multiplied by the number of concurrent workers. Each worker requires its own thread/process and a good rule of thumb is to configure not more than twice as many workers as the number of cores available on that host. Otherwise you will see a decrease in overall performance, caused by too many context switches, for which the scheduler of the operating system is responsible.

Due to this workflow, it is almost impossible to add support for long term connections, such as websockets, on top of the WSGI protocol specification. Therefore most websocket implementations go for another approach. The websocket connection is controlled by a service running side by side with the default application server. Here, a webserver with support for long term connections, dispatches the requests from the clients.

A webserver able to dispatch websocket requests is the NGiNX server. Normal requests are sent to Django using the WSGI protocol, whereas the long living websocket connections are passed over to a special service responsible only for that.

A typical implementation proposal is to use socket.io running inside a NodeJS loop.



Here, **Django** communicates with **Node.JS** using a RESTful API. This however is hard to maintain because it pulls in two completely different technologies. In alternative proposals, other Python based asynchronous event frameworks such as Tornado or Twisted are used. But they all look like makeshift solutions, since one has to run a second framework side by side with **Django**. This makes the project dependent on another infrastructure and thus harder to maintain. Moreover, having to run two concurrent frameworks can be quite embarrassing during application development, specially while debugging code.

uWSGI

While searching for a simpler solution, I found out that uWSGI offers websockets right out of the box. With Redis as a message queue, and a few lines of Python code, one can bidirectionally communicate with any WSGI based framework, for instance **Django**. Of course, here it also is prohibitive to create a new thread for each open websocket connection. Therefore that part of the code runs in one single thread/process for all open connections in a cooperative concurrency mode using the excellent gevent and greenlet libraries.

This approach has some advantages:

- It is simpler to implement.
- **The asynchronous I/O loop handling websockets can run**
  - inside Django with `./manage.py runserver`, giving full debugging control.
  - as a stand alone HTTP server, using uWSGI.
  - using NGiNX or Apache (>= 2.4) as proxy in two decoupled loops, one for WSGI and one for websocket HTTP in front of two separate uWSGI workers.

- The whole Django API is available in this loop, provided that no blocking calls are made. Therefore the websocket code can access the Django configuration, the user and the session cache, etc.

Using Redis as a message queue

One might argue that all this is not as simple, since an additional service – the Redis data server – must run side by side with Django. Websockets are bidirectional but their normal use case is to trigger server initiated events on the client. Although the other direction is possible, it can be handled much easier using Ajax – adding an additional TCP/IP handshake.

Here, the only "stay in touch with the client" is the file descriptor attached to the websocket. And since we speak about thousands of open connections, the footprint in terms of memory and CPU resources must be brought down to a minimum. In this implementation, only one open file handle is required for each open websocket connection.

Productive webservers require some kind of session store anyway. This can be a memcached or a Redis data server. Therefore, such a service must run anyway and if we can choose between one of them, we shall use one with integrated message queuing support. When using Redis for caching and as a session store, we practically get the message queue for free.

Scalability

One of the nice features of Redis is its infinite scalability. If one Redis server can't handle its workload, interconnect it with another one and all events and messages are mirrored across this network. Since **django-websocket-redis** can be deployed multiple times and as self-contained Django applications, this configuration can scale infinitely, just interconnect the Redis servers to each other.

On the main entry point of your site, add a loadbalancer capable of proxying the websocket protocol. This can be any OSI level 4 loadbalancer such as the Linux Virtual Server project, or if you prefer OSI level 7, the excellent HAProxy.

Installation and Configuration

Installation

If not already done, install the **Redis server**, using the installation tool offered by the operating system, such as `aptitude`, `yum`, `port` or install Redis from source.

Start the Redis service on your host

```
$ sudo service redis-server start
```

Check if Redis is up and accepting connections

```
$ redis-cli ping
PONG
```

Install **Django Websocket for Redis**. The latest stable release can be found on PyPI

```
pip install django-websocket-redis
```

or the newest development version from github

```
pip install -e git+https://github.com/jrief/django-websocket-redis#egg=django-
↪websocket-redis
```

Websocket for Redis does not define any database models. It can therefore be installed without any database synchronization.

Dependencies

- Django >=1.5
- redis >=2.10.3 (a Python client for Redis)
- uWSGI >=1.9.20
- gevent >=1.0.1
- greenlet >=0.4.5
- optional, but recommended: wsaccel >=0.6

Configuration

Add `"ws4redis"` to your project's `INSTALLED_APPS` setting

```
INSTALLED_APPS = (
    ...
    'ws4redis',
    ...
)
```

Specify the URL that distinguishes websocket connections from normal requests

```
WEBSOCKET_URL = '/ws/'
```

If the Redis datastore uses connection settings other than the defaults, use this dictionary to override these values

```
WS4REDIS_CONNECTION = {
    'host': 'redis.example.com',
    'port': 16379,
    'db': 17,
    'password': 'verysecret',
}
```

Note: Specify only the values, which deviate from the default.

If your Redis instance is accessed via a Unix Domain Socket, you can configure that as well:

```
WS4REDIS_CONNECTION = {
    'unix_socket_path': '/tmp/redis.sock',
    'db': 5
}
```

Websocket for Redis can be configured with `WS4REDIS_EXPIRE`, to additionally persist messages published on the message queue. This is advantageous in situations, where clients shall be able to access the published information after reconnecting the websocket, for instance after a page is reloaded.

This directive sets the number in seconds, each received message is persisted by Redis, additionally of being published on the message queue

```
WS4REDIS_EXPIRE = 7200
```

Websocket for Redis can prefix each entry in the datastore with a string. By default, this is empty. If the same Redis connection is used to store other kinds of data, in order to avoid name clashes you're encouraged to prefix these entries with a unique string, say

```
WS4REDIS_PREFIX = 'ws'
```

Override `ws4redis.store.RedisStore` with a customized class, in case you need an alternative implementation of that class

```
WS4REDIS_SUBSCRIBER = 'myapp.redis_store.RedisSubscriber'
```

This directive is required during development and ignored in production environments. It overrides Django's internal main loop and adds a URL dispatcher in front of the request handler

```
WSGI_APPLICATION = 'ws4redis.django_runserver.application'
```

Ensure that your template context contains at least these processors:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'django.contrib.auth.context_processors.auth',
    'django.core.context_processors.static',
    'ws4redis.context_processors.default',
    ...
)
```

Websocket for Redis allows each client to subscribe and to publish on every possible channel. To restrict and control access, the `WS4REDIS_ALLOWED_CHANNELS` options should be set to a callback function anywhere inside your project. See the example and warnings in *Safety considerations*.

Check your Installation

With **Websockets for Redis** your Django application has immediate access to code written for websockets. Change into the `examples` directory and start a sample chat server

```
./manage.py migrate
... create database tables
... answer the questions
./manage.py runserver
```

Point a browser onto http://localhost:8000/chat/, you should see a simple chat server. Enter a message and send it to the server. It should be echoed immediately on the billboard.

Point a second browser onto the same URL. Now each browser should echo the message entered into input field.

In the examples directory, there are two chat server implementations, which run out of the box. One simply broadcasts messages to every client listening on that same websocket URL. The other chat server can be used to send messages to specific users logged into the system. Use these demos as a starting point for your application.

Replace memcached with Redis

Since Redis has to be added as an additional service to the current infrastructure, at least another service can be safely removed: *memcached*. This is required by typical Django installations and is used for caching and session storage.

It's beyond the scope of this documentation to explain how to set up a caching and/or session store using Redis, so please check django-redis-sessions and optionally django-redis-cache for details, but it should be as easy as installing

```
pip install django-redis-sessions
```

and adding

```
SESSION_ENGINE = 'redis_sessions.session'
SESSION_REDIS_PREFIX = 'session'
```

to the file `settings.py`. Here is a full description on how to use Redis as Django session store and cache backend.

Also keep in mind, that accessing session data is a blocking I/O call. Hence the connection from the websocket loop to the session store **must use gevent**, otherwise the websockets may block altogether. Therefore, if you for some reason you have to remain with your current session store, make sure its monkey patched with gevent.

Warning: **Never** store session data in the database in combination with *Websockets for Redis*!

Running WebSocket for Redis

WebSocket for Redis is a library which runs side by side with Django. It has its own separate main loop, which does nothing else than keeping the WebSocket alive and dispatching requests from **Redis** to the configured WebSockets and vice versa.

Django with WebSockets for Redis in development mode

With **WebSockets for Redis**, a Django application has immediate access to code written for WebSockets. Make sure, that Redis is up and accepts connections.

```
$ redis-cli ping
PONG
```

Then start the Django development server.

```
./manage.py runserver
```

As usual, this command shall only be used for development.

The `runserver` command is a monkey patched version of the original Django main loop and works similar to it. If an incoming request is of type WSGI, everything works as usual. However, if the patched handler detects an incoming request wishing to open a WebSocket, then the Django main loop is hijacked by **ws4redis**. This separate loop then waits until `select` notifies that some data is available for further processing, or by the WebSocket itself, or by the Redis message queue. This hijacked main loop finishes when the WebSocket is closed or when an error occurs.

Note: In development, one thread is created for each open WebSocket.

Opened WebSocket connections exchange so called Ping/Pong messages. They keep the connections open, even if there is no payload to be sent. In development mode, the "WebSocket" main loop does not send these stay alive packages, because normally there is no proxy or firewall between the server and the client which could drop the connection. This could be easily implemented, though.

Django with WebSockets for Redis as a stand alone uWSGI server

In this configuration the **uWSGI** server owns the main loop. To distinguish WebSockets from normals requests, modify the Python starter module `wsgi.py` to

```
import os
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myapp.settings')
from django.conf import settings
from django.core.wsgi import get_wsgi_application
from ws4redis.uwsgi_runserver import uWSGIWebsocketServer

_django_app = get_wsgi_application()
_websocket_app = uWSGIWebsocketServer()

def application(environ, start_response):
    if environ.get('PATH_INFO').startswith(settings.WEBSOCKET_URL):
        return _websocket_app(environ, start_response)
    return _django_app(environ, start_response)
```

Run uWSGI as stand alone server with

```
uwsgi --virtualenv /path/to/virtualenv --http :80 --gevent 100 --http-websockets --
↪module wsgi
```

This will answer, both Django and WebSocket requests on port 80 using HTTP. Here the modified `application` dispatches incoming requests depending on the URL on either a Django handler or into the WebSocket's main loop.

This configuration works for testing uWSGI and low traffic sites. Since uWSGI then runs in one thread/process, blocking calls such as accessing the database, would also block all other HTTP requests. Adding `--gevent-monkey-patch` to the command line may help here, but Postgres for instance requires to monkey patch its blocking calls with **gevent** using the psycogreen library. Moreover, only one CPU core is then used, and static files must be handled by another webserver.

Serving static files

In this configuration, you are not able to serve static files, because Django does not run in debug mode and uWSGI does not know how to server your deployed static files. Therefore in `urls.py` add `staticfiles_urlpatterns` to your urlpatterns:

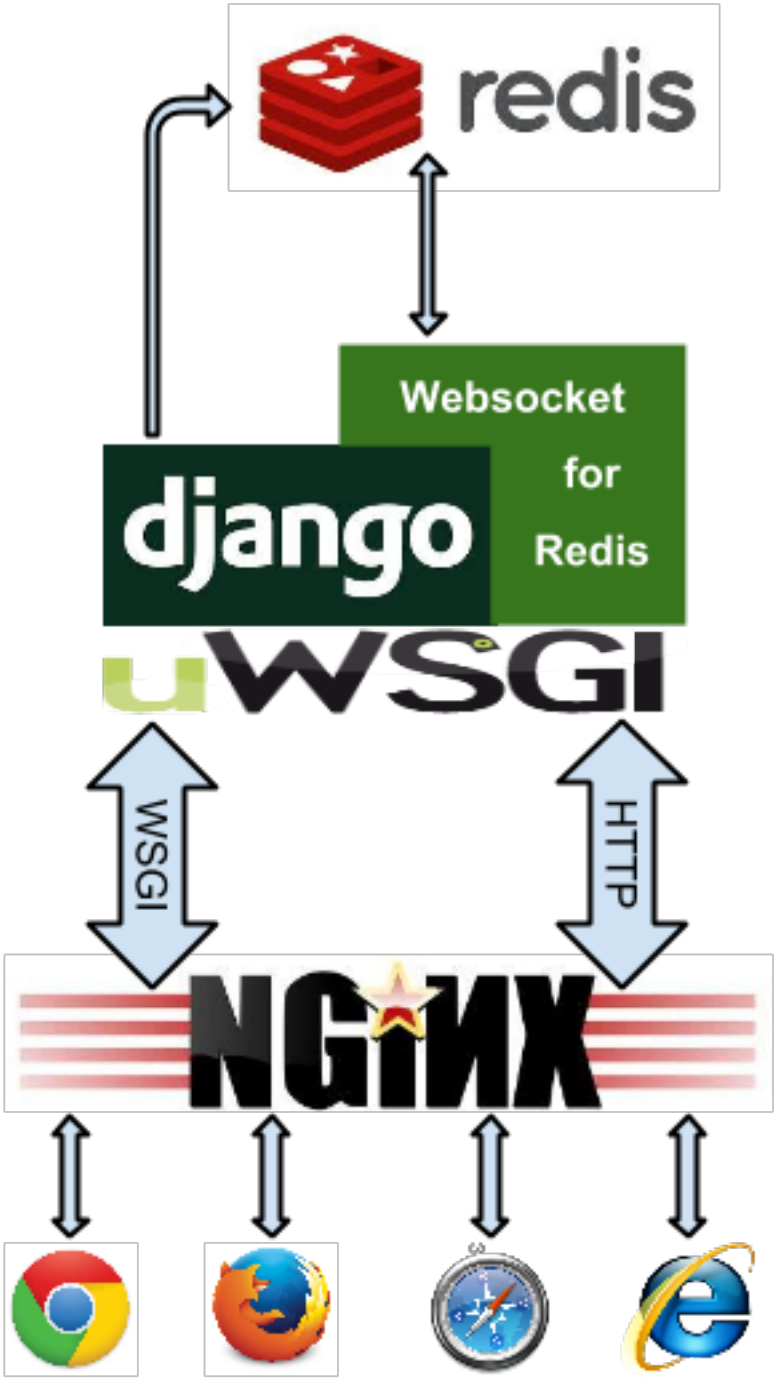
```
from django.conf.urls import url, patterns, include
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

urlpatterns = patterns('',
    ....
) + staticfiles_urlpatterns()
```

Note: Remember to remove `staticfiles_urlpatterns` when upgrading to a more scalable configuration as explained in the next section.

Django with WebSockets for Redis behind NGiNX using uWSGI

This is the most scalable solution. Here two instances of a uWSGI server are spawned, one to handle normal HTTP requests for Django and one to handle WebSocket requests.



Assure that you use NGiNX version 1.3.13 or later, since earlier versions have no support for WebSocket proxying. The web server undertakes the task of dispatching normal requests to one uWSGI instance and WebSocket requests to another one. The responsible configuration section for NGiNX shall look like:

```
location / {
    include /etc/nginx/uwsgi_params;
    uwsgi_pass unix:/path/to/django.socket;
}
```

```
location /ws/ {
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_pass http://unix:/path/to/web.socket;
}
```

For details refer to NGiNX's configuration on WebSocket proxying.

Since both uWSGI handlers create their own main loop, they also require their own application and different UNIX sockets. Create two adopter files, one for the Django loop, say `wsgi_django.py`

```
import os
os.environ.update(DJANGO_SETTINGS_MODULE='my_app.settings')
from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

and one for the WebSocket loop, say `wsgi_websocket.py`

```
import os
import gevent.socket
import redis.connection
redis.connection.socket = gevent.socket
os.environ.update(DJANGO_SETTINGS_MODULE='my_app.settings')
from ws4redis.uwsgi_runserver import uWSGIWebsocketServer
application = uWSGIWebsocketServer()
```

Start those two applications as separate uWSGI instances

```
uwsgi --virtualenv /path/to/virtualenv --socket /path/to/django.socket --buffer-
↪size=32768 --workers=5 --master --module wsgi_django
uwsgi --virtualenv /path/to/virtualenv --http-socket /path/to/web.socket --gevent␣
↪1000 --http-websockets --workers=2 --master --module wsgi_websocket
```

The NGiNX web server is now configured as a scalable application server which can handle a thousand WebSockets connections concurrently.

If you feel uncomfortable with separating WebSocket from normal requests on NGiNX, consider that you already separate static and media requests on the web server. Hence, WebSockets are just another extra routing path.

Django with WebSockets for Redis behind Apache-2.4 using uWSGI

Mike Martinka <mike.martinka@ntrepidcorp.com> reported this configuration, which allows to run **ws4redis** with Apache-2.4 and later.

Configuratin for uWSGI:

```
[uwsgi]
env=DJANGO_SETTINGS_MODULE=<app>.settings
module=<module>:application
master=True
http-socket=127.0.0.1:9090
http-websockets=true
gevent=1000
workers=2
plugin=python
```

Configuration section for Apache:

```
<VirtualHost IPADDR:80>
    ProxyPass    /ws/    ws://127.0.0.1:9090/
</VirtualHost>
```

Django with WebSockets for Redis as a stand alone uWSGI server in emperor mode

In this configuration the **uWSGI** server owns both main loops. To distinguish WebSockets from normal requests, use uWSGI's internal routing capabilities.

Note: The internal routing capabilities of uWSGI is dependent on the Perl Compatible Regular Expressions (PCRE) library. Make sure that your uWSGI was built with PCRE support if you plan to run in emperor mode. Please refer to the *PCRE Support* section below for more information.

First create the two applications, `wsgi_django.py` and `wsgi_websocket.py` using the same code as in the above example. These are the two entry points for uWSGI. Then create these three ini-files, one for the emperor, say `uwsgi.ini`:

```
[uwsgi]
emperor = vassals
http-socket = :9090
die-on-term = true
offload-threads = 1
route = ^/ws uwsgi:/var/tmp/web.socket,0,0
route = ^/ uwsgi:/var/tmp/django.socket,0,0
```

Create a separate directory named `vassals` and add a configuration file for the Websocket loop, say `vassals/wsserver.ini`:

```
; run the Websocket loop
[uwsgi]
umask = 002
virtualenv = /path/to/your/virtualenv
chdir = ..
master = true
no-orphans = true
die-on-term = true
memory-report = true
env = DJANGO_SETTINGS_MODULE=my_app.settings
socket = /var/tmp/web.socket
module = wsgi_websocket:application
threads = 1
processes = 1
http-websockets = true
gevent = 1000
```

To the directory named `vassals`, add a configuration file for the Django loop, say `vassals/runserver.ini`:

```
; run the Django loop
[uwsgi]
umask = 002
virtualenv = /path/to/your/virtualenv
chdir = ..
master = true
```

```
no-orphans = true
die-on-term = true
memory-report = true
env = DJANGO_SETTINGS_MODULE=my_app.settings
socket = /var/tmp/django.socket
module = wsgi_django:application
buffer-size = 32768
threads = 1
processes = 2
```

Adopt the virtualenv, pathes, ports and number of threads/processes to your operating system and hosts capabilities.

Then start uWSGI:

```
uwsgi --ini uwsgi.ini
```

This configuration scales as well, as the sample from the previous section. It shall be used if no NGiNX server is available.

Serving static files

The alert reader will have noticed, that static files are not handled by this configuration. While in theory it is possible to configure **uWSGI** to deliver static files, please note that **uWSGI** is not intended to completly replace a webserver. Therefore, before adding `route = ^/static static:/path/to/static/root` to the emperors ini-file, consider to place them onto a Content Delivery Network, such as Amazon S3.

PCRE Support

If you encounter the error message `!!! no internal routing support, rebuild with pcre support !!!` in the logs/console when running in emperor mode, that means you were lacking the PCRE libraries when you first installed uWSGI. You will need to rebuild the uWSGI binaries. To do that uninstall uWSGI, and then download the `libpcre3` and `libpcre3-dev` libraries using your system's package management tool. Once finished, reinstall uWSGI. Credits to this post.

Using Websockets for Redis

Websocket for Redis allows uni- and bidirectional communication from the client to the server and vice versa. Each websocket is identified by the part of the URL which follows the prefix `/ws/`. Use different uniform locators to distinguish between unrelated communication channels.

Note: The prefix `/ws/` is specified using the configuration setting `WEBSOCKET_URL` and can be changed to whatever is appropriate.

Client side

The idea is to let a client subscribe for different channels, so that he only gets notified, when a certain event happens on a channel he is interested in. Currently there are four such events, *broadcast notification*, *user notification*, *group notification* and *session notification*. Additionally, a client may declare on initialization, on which channels he wishes to publish a message. The latter is not that important for a websocket implementation, because it can be achieved otherwise, using the well known XMLHttpRequest (Ajax) methods.

A minimal client in pure JavaScript

```
var ws = new WebSocket('ws://www.example.com/ws/foobar?subscribe-broadcast&publish-
↪broadcast&echo');
ws.onopen = function() {
    console.log("websocket connected");
};
ws.onmessage = function(e) {
    console.log("Received: " + e.data);
};
ws.onerror = function(e) {
    console.error(e);
};
ws.onclose = function(e) {
    console.log("connection closed");
}
function send_message(msg) {
    ws.send(msg);
}
```

Client JavaScript depending on jQuery (recommended)

When using jQuery, clients can reconnect on broken Websockets. Additionally the client awaits for heartbeat messages and reconnects if too many of them were missed.

Include the client code in your template:

```
<script type="text/javascript" src="{{ STATIC_URL }}js/ws4redis.js"></script>
```

and access the Websocket code:

```
jQuery(document).ready(function($) {
    var ws4redis = WS4Redis({
        uri: '{{ WEBSOCKET_URI }}foobar?subscribe-broadcast&publish-broadcast&echo',
        connecting: on_connecting,
        connected: on_connected,
        receive_message: receiveMessage,
        disconnected: on_disconnected,
        heartbeat_msg: {{ WS4REDIS_HEARTBEAT }}
    });

    // attach this function to an event handler on your site
    function sendMessage() {
        ws4redis.send_message('A message');
    }

    function on_connecting() {
        alert('Websocket is connecting...');
    }

    function on_connected() {
        ws4redis.send_message('Hello');
    }

    function on_disconnected(evt) {
        alert('Websocket was disconnected: ' + JSON.stringify(evt));
    }
```

```
    // receive a message though the websocket from the server
    function receiveMessage(msg) {
        alert('Message from Websocket: ' + msg);
    }
});
```

If you want to close the connection explicitly, you could call **ws4redis.close()**. This way, the client will not perform reconnection attempts.

This example shows how to configure a Websocket for bidirectional communication.

Note: A client wishing to trigger events on the server side, shall use XMLHttpRequests (Ajax), as they are much more suitable, rather than messages sent via Websockets. The main purpose for Websockets is to communicate asynchronously from the server to the client.

Server Side

The Django loop is triggered by client HTTP requests, except for special cases such as jobs triggered by, for instance django-celery. Intentionally, there is no way to trigger events in the Django loop through a Websocket request. Hence, all of the communication between the Websocket loop and the Django loop must pass through the message queue.

RedisSubscriber

In the Websocket loop, the message queue is controlled by the class `RedisSubscriber`, which can be replaced using the configuration directive `WS4REDIS_SUBSCRIBER`.

RedisPublisher

In the Django loop, this message queue is controlled by the class `RedisPublisher`, which can be accessed by any Django view.

Both, `RedisSubscriber` and `RedisPublisher` share the same base class `RedisStore`.

Subscribe to Broadcast Notifications

This is the simplest form of notification. Every Websocket subscribed to a broadcast channel is notified, when a message is sent to that named Redis channel. Say, the Websocket URL is `ws://www.example.com/ws/foobar?subscribe-broadcast` and the Django loop wants to publish a message to all clients listening on the named facility, referred here as `foobar`.

```
from ws4redis.publisher import RedisPublisher
from ws4redis.redis_store import RedisMessage

redis_publisher = RedisPublisher(facility='foobar', broadcast=True)
message = RedisMessage('Hello World')
# and somewhere else
redis_publisher.publish_message(message)
```

now, the message “Hello World” is received by all clients listening for that broadcast notification.

Subscribe to User Notification

A Websocket initialized with the URL `ws://www.example.com/ws/foobar?subscribe-user`, will be notified if that connection belongs to a logged in user and someone publishes a message on for that user, using the `RedisPublisher`.

```
redis_publisher = RedisPublisher(facility='foobar', users=['john', 'mary'])
message = RedisMessage('Hello World')
# and somewhere else
redis_publisher.publish_message(message)
```

now, the message "Hello World" is sent to all clients logged in as `john` or `mary` and listening for that kind of notification.

If the message shall be send to the currently logged in user, then you may use the magic item `SELF`.

```
from ws4redis.redis_store import SELF

redis_publisher = RedisPublisher(facility='foobar', users=[SELF], request=request)
```

Subscribe to Group Notification

A Websocket initialized with the URL `ws://www.example.com/ws/foobar?subscribe-group`, will be notified if that connection belongs to a logged in user and someone publishes a message for a group where this user is member of.

```
redis_publisher = RedisPublisher(facility='foobar', groups=['chatters'])

# and somewhere else
redis_publisher.publish_message('Hello World')
```

now, the message "Hello World" is sent to all clients logged in as users which are members of the group `chatters` and subscribing to that kind of notification.

In this context the the magic item `SELF` refers to all the groups, the current logged in user belongs to.

Note: This feature uses a signal handler in the Django loop, which determines the groups a user belongs to. This list of groups then is persisted inside a session variable to avoid having the Websocket loop to access the database.

Subscribe to Session Notification

A Websocket initialized with the URL `ws://www.example.com/ws/foobar?subscribe-session`, will be notified if someone publishes a message for a client owning this session key.

```
redis_publisher = RedisPublisher(facility='foobar', sessions=[
↪'wnqd0gbw5obpnj50zwh6yaq2yz4o8g9x'])
message = RedisMessage('Hello World')

# and somewhere else
redis_publisher.publish_message(message)
```

now, the message "Hello World" is sent to all clients using the session key `wnqd0gbw5obpnj50zwh6yaq2yz4o8g9x` and subscribing to that kind of notification.

In this context the the magic item `SELF` refers to all clients owning the same session key.

Publish for Broadcast, User, Group and Session

A Websocket initialized with the URL `ws://www.example.com/ws/foobar?publish-broadcast`, `ws://www.example.com/ws/foobar?publish-user` or `ws://www.example.com/ws/foobar?publish-session` will publish a message sent through the Websocket on the named Redis channel `broadcast:foobar`, `user:john:foobar` and `session:wnqd0gbw5obpnj50zwh6yaq2yz4o8g9x:foobar` respectively. Every listener subscribed to any of the named channels, then will be notified.

This configuration only makes sense, if the messages send by the client using the Websocket, shall not trigger any server side event. A practical use would be to store current the GPS coordinates of a moving client inside the Redis datastore. Then Django can fetch these coordinates from Redis, whenever it requires them.

```
# if the publisher is required only for fetching messages, use an
# empty constructor, otherwise reuse an existing redis_publisher
redis_publisher = RedisPublisher()

# and somewhere else
facility = 'foobar'
audience = 'any'
redis_publisher.fetch_message(request, facility, audience)
```

The argument `audience` must be one of `broadcast`, `group`, `user`, `session` or `any`. The method `fetch_message` searches through the Redis datastore to find a persisted message for that channel. The first found message is returned to the caller. If no matching message was found, `None` is returned.

Message echoing

Some kind of applications require to just hold a state object on the server-side, which is a copy of a corresponding JavaScript object on the client. These applications do not require message echoing. Here an incoming message is only dispatched to the subscribed websockets, if the this message contains a different content. This is the default setting.

Other applications such as chats or games, must be informed on each message published on the message queue, regardless of its content. These applications require message echoing. Here an incoming message is always dispatched to the subscribed websockets. To activate message echoing, simply append the parameter `&echo` to the URL used for connecting to the websocket.

Persisting messages

If a client connects to a Redis channel for the first time, or if he reconnects after a page reload, he might be interested in the current message, previously published on that channel. If the configuration settings `WS4REDIS_EXPIRE` is set to a positive value, **Websocket for Redis** persists the current message in its key-value store. This message then is retrieved and sent to the client, immediately after he connects to the server.

Note: By using client code, which automatically reconnects after the Websocket closes, one can create a setup which is immune against server and client reboots.

Safety considerations

The default setting of **Websocket for Redis** is to allow each client to subscribe and to publish on every possible channel. This normally is not what you want. Therefore **Websocket for Redis** allows to restrict the channels for subscription and publishing to your application needs. This is done by a callback function, which is called right after

the initialization of the Websocket. This function shall be used to restrict the subscription/publishing channels for the current client.

Example:

```
def get_allowed_channels(request, channels):
    return set(channels).intersection(['subscribe-broadcast', 'subscribe-group'])
```

This function restricts the allowed channels to `subscribe-broadcast` and `subscribe-group` only. All other attempts to subscribe or to publish on other channels will be silently discarded.

Disallow non authenticated users to subscribe or to publish on the Websocket:

```
from django.core.exceptions import PermissionDenied

def get_allowed_channels(request, channels):
    if not request.user.is_authenticated():
        raise PermissionDenied('Not allowed to subscribe nor to publish on the␣
↪Websocket!')
```

When using this callback function, Websockets opened by a non-authenticated users, will get a **403 - Response Forbidden** error.

To enable this function in your application, use the configuration directive `WS4REDIS_ALLOWED_CHANNELS`.

Note: This function must not perform any blocking requests, such as accessing the database!

Sending and receiving heartbeat messages

The Websocket protocol implements so called PING/PONG messages to keep Websockets alive, even behind proxies, firewalls and load-balancers. The server sends a PING message to the client through the Websocket, which then replies with PONG. If the client does not reply, the server closes the connection.

The client part

Unfortunately, the Websocket protocol does not provide a similar method for the client, to find out if it is still connected to the server. This can happen, if the connection simply disappears without further notification. In order to have the client recognize this, some Javascript code has to be added to the client code responsible for the Websocket:

```
var ws = new WebSocket('ws://www.example.com/ws/foobar?subscribe-broadcast');
var heartbeat_msg = '--heartbeat--', heartbeat_interval = null, missed_heartbeats = 0;

function on_open() {
    // ...
    // other code which has to be executed after the client
    // connected successfully through the websocket
    // ...
    if (heartbeat_interval === null) {
        missed_heartbeats = 0;
        heartbeat_interval = setInterval(function() {
            try {
                missed_heartbeats++;
                if (missed_heartbeats >= 3)
                    throw new Error("Too many missed heartbeats.");
```

```
                ws.send(heartbeat_msg);
            } catch(e) {
                clearInterval(heartbeat_interval);
                heartbeat_interval = null;
                console.warn("Closing connection. Reason: " + e.message);
                ws.close();
            }
        }, 5000);
    }
}
```

The heartbeat message, here `--heartbeat--` can be any magic string which does not interfere with your remaining logic. The best way to achieve this, is to check for that magic string inside the receive function, just before further processing the message:

```
function on_message(evt) {
    if (evt.data === heartbeat_msg) {
        // reset the counter for missed heartbeats
        missed_heartbeats = 0;
        return;
    }
    // ...
    // code to further process the received message
    // ...
}
```

The server part

The main loop of the Websocket server is idle for a maximum of 4 seconds, even if there is nothing to do. After that time interval has elapsed, this loop optionally sends a magic string to the client. This can be configured using the special setting:

```
WS4REDIS_HEARTBEAT = '--heartbeat--'
```

The purpose of this setting is twofold. During processing, the server ignores incoming messages containing this magic string. Additionally the Websocket server sends a message with that magic string to the client, about every four seconds. The above client code awaits these messages, at least every five seconds, and if too many were not received, it closes the connection and tries to reestablish it.

By default the setting `WS4REDIS_HEARTBEAT` is `None`, which means that heartbeat messages are neither expected nor sent.

Application Programming Interface

This document describes how to interact with **Websockets for Redis** from the Django loop and how to adopt the Websocket loop for other purposes.

Use `RedisPublisher` from inside Django views

For obvious architectural reasons, the code handling the websocket loop can not be accessed directly from within Django. Therefore, all communication from Django to the websocket loop, must be passed over to the Redis message queue and vice versa. To facility this, **ws4redis** offers a class named `RedisPublisher`. An instance of this class

shall be used from inside Django views to push messages via a websocket to the client, or to fetch persisted messages sent through the websocket.

Example view:

```
from django.views.generic.base import View
from ws4redis.publisher import RedisPublisher

class MyTypicalView(View):
    facility = 'unique-named-facility'
    audience = {'broadcast': True}

    def __init__(self, *args, **kwargs):
        super(MyTypicalView, self).init(*args, **kwargs)
        self.redis_publisher = RedisPublisher(facility=self.facility, **self.audience)

    def get(self, request)
        message = 'A message passed to all browsers listening on the named facility'
        self.redis_publisher.publish_message(message)
```

For further options, refer to the reference:

`RedisStore.`**`publish_message`**(*message*, *expire=None*)
Publish a `message` on the subscribed channel on the Redis datastore. `expire` sets the time in seconds, on how long the message shall additionally of being published, also be persisted in the Redis datastore. If unset, it defaults to the configuration settings `WS4REDIS_EXPIRE`.

Replace `RedisSubscriber` for the Websocket loop

Sometimes the predefined channels for subscribing and publishing messages might not be enough. If there is a need to add additional channels to the message queue, it is possible to replace the implemented class `ws4redis.store.RedisSubscriber` by setting the configuration directive `WS4REDIS_SUBSCRIBER` to a class of your choice.

Use the class `RedisSubscriber` as a starting point and overload the required methods with your own implementation.

class `ws4redis.subscriber.`**`RedisSubscriber`**(*connection*)
Subscriber class, used by the websocket code to listen for subscribed channels

`get_file_descriptor`()
Returns the file descriptor used for passing to the select call when listening on the message queue.

`parse_response`()
Parse a message response sent by the Redis datastore on a subscribed channel.

`release`()
New implementation to free up Redis subscriptions when websockets close. This prevents memory sap when Redis Output Buffer and Output Lists build when websockets are abandoned.

`send_persited_messages`(*websocket*)
This method is called immediately after a websocket is openend by the client, so that persisted messages can be sent back to the client upon connection.

`set_pubsub_channels`(*request*, *channels*)
Initialize the channels used for publishing and subscribing messages through the message queue.

Warning: If the overloaded class calls any blocking functions, such as `sleep`, `read`, `select` or similar, make sure that these functions are patched by the gevent library, otherwise *all* connections will block simultaneously.

Testing Websockets for Redis

A simple Chat server

In the `examples` directory, there are two demo chat servers. To start them, first initialize the SQLite database

```
# create python2 virtualenv
virtualenv - p /path/to/python2 /path/to/virtualenv

# activate virtualenv
source /path/to/virtualenv/bin/activate

# Make sure you're in the examples/ directory
cd examples/

# install pip requirements
pip install -r requirements.txt

# Django 1.7+
# Load test data
./manage.py migrate
./manage.py loaddata chatserver/fixtures/data.json
```

and then start the server

```
# start Redis Server from a different shell prompt
# (or follow quickstart instructions http://redis.io/topics/quickstart)
redis-server

# start Django
./manage.py runserver
```

Point a browser onto http://localhost:8000/admin/, login as the 'admin' user using the password 'secret' and add additional users. Enable their staff status, so that they can use the admin interface to log into the testing application.

With http://localhost:8000/chat/ you can send messages to specific users, provided they are logged in. To log in as another user, use Django's admin interface.

Simple Broadcasting

On http://localhost:8000/chat/ there is a chat server, which simply broadcasts messages to all browsers accessing this same URL.

Testing uWSGI

Before configuring NGiNX to run in front of two instances of uWSGI, it is recommended to run uWSGI as a stand alone server for testing purpose. The entry point of this server makes the distinction between normal HTTP and Websocket requests. In directory `examples`, start uwsgi as

```
uwsgi --virtualenv /path/to/virtualenvs --http :9090 --gevent 100 --http-websockets --
↪module wsgi
```

Both chat server tests from above should run in this configuration.

Running Unit Tests

```
./manage.py test chatserver --settings=chatserver.tests.settings
```

Currently it is not possible to simulate more than one client at a time. Django's built in LiveServerTestCase can not handle more than one simultaneous open connection, and thus more sophisticated tests with more than one active Websockets are not possible.

Running Stress Tests

To run stress tests, change into directory `stress-tests`. Since stress tests shall check the performance in a real environment, the server and the testing client must be started independently.

First start the server, as you would in productive environments.

```
# Open a new shell and activate your virtualenv in it
source /path/to/virtualenv/bin/activate

# Install the uwsgi package
pip install uwsgi

# Then start the uwsgi server
uwsgi --http :8000 --gevent 1000 --http-websockets --master --workers 2 --module wsgi_
↪websocket
```

then go back to the other shell (also with the virtualenv activated) and start one of the testing clients, using the nose framework

```
nosetests test_uwsgi_gevent.py
```

(this test, on my MacBook, requires about 1.5 seconds)

or start a similar test using real threads instead of greenlets

```
nosetests test_uwsgi_threads.py
```

(this test, on my MacBook, requires about 2.5 seconds)

Both clients subscribe to 1000 concurrent Websockets. Then a message is published from another Websocket. If all the clients receive that message, the test is considered as successful. Both perform the same test, but `test_uwsgi_gevent.py` uses greenlet's for each client to simulate, whereas `test_uwsgi_threads.py` uses Python thread's.

If these tests do not work in your environment, check your file descriptors limitations. Use the shell command `ulimit -n` and adopt it to these requirements. Alternatively reduce the number of concurrent clients in the tests.

Debugging

This project adds some extra complexity to Django projects with websocket-redis. This is because now there are two entry points instead of one. The default **Django** one, based on the WSGI protocol, which is used to handle the typical HTTP-Request-Response. And the new one **Websocket for Redis**, based on the HTTP, which handles the websocket part.

Django Loop and Websocket Loop

In this documentation, I use the terms *Django Loop* and *Websocket Loop* to distinguish these two entry points. You shall rarely need to access the Websocket Loop, because intentionally there are no hooks for adding server side logics. The latter must reside inside the Django loop using Redis as the communication engine between those two.

A reason one might need to debug inside the Websocket loop, is, because the subscriber was overridden using the configuration setting `WS4REDIS_SUBSCRIBER`. Therefore, one of the aims of this project is to facilitate the entry level for debugging. During development, hence the server is started with `./manage.py runserver`, this is achieved by hijacking the Django loop. Then the connection is kept open, until the client closes the Websocket.

If existing workers do not return, Django creates a thread for new incoming requests. This means that during debugging, each Websocket connection owns its own thread. Such an approach is perfectly feasible, however it scales badly and therefore should not be used during production.

Query the datastore

Sometimes you might need to know, why some data is bogus or was not sent/received by the client. The easiest way to do this is to access the Redis datastore.

```
$ redis-cli
redis 127.0.0.1:6379>
```

In this command line interface, you can find out about all the data managed by **Websocket for Redis**. Redis offers many commands from which a few are useful here:

keys

```
redis 127.0.0.1:6379> keys *
```

Gives a list of all keys used in Redis. If a `WS4REDIS_PREFIX` is specified in `settings.py`, this prefixing string can be used to limit the keys to those used by **Websocket for Redis**.

If, for instance you're interested into all messages available for broadcast, then invoke:

```
redis 127.0.0.1:6379> keys [prefix:]broadcast:*
```

with the *prefix*, if set.

get

```
redis 127.0.0.1:6379> get [prefix:]broadcast:foo
```

This returns the data available for broadcast for the facility named "foo".

```
redis 127.0.0.1:6379> get [prefix:]user:john:foo
```

This returns the data available for user "john" for the facility named "foo".

```
redis 127.0.0.1:6379> get [prefix:]session:wnqd0gbw5obpnj50zwh6yaq2yz4o8g9x:foo
```

This returns the data available for the browser owning the session-id `wnqd0gbw5obpnj50zwh6yaq2yz4o8g9x` for the facility named "foo".

subscribe

If **Websocket for Redis** is configured to not cache published data, no data buckets are filled. This is the case, when the configuration option `WS4REDIS_EXPIRE` is set to zero or None. In such a situation, the Redis commands `keys` and `get` won't give you any information. But you can subscribe for listening to a named channel:

```
redis 127.0.0.1:6379> subscribe [prefix:]broadcast:foo
```

This command blocks until some data is received. It then dumps the received data.

You have to reenter the subscribe command, if you want to listen for further data.

Release History

0.5.0

- Support for Django-1.11.

0.4.8

- Support Redis connections over Unix Domain Sockets.

0.4.7

Improvements to the javascript API:

- Performing reconnection attempts when the first connection (on instantiation) fails.
- Adding the 'close()' method to enable closing the connection explicitly. When the connection is closed calling this method, there will be no reconnection attempts. In order to connect again, the client must be re-instantiated.
- Adding 'connecting' and 'disconnected' callback options. The first is fired right before the Websocket is being instantiated, while tha last is fired after connection is closed.
- Adding the following methods to check websocket status: `is_connecting()`, `is_connected()`, `is_closing()`, `is_closed()`.
- Replaced `STATIC_URL` against `{% static %}` in all templates.
- Keep track on opened websockets.

0.4.6

- Added support for the Sec-WebSocket-Protocol header. Thanks to Erwin Junge.
- Fixed bug in unpacking binary websocket protocol.

0.4.5

- created 1 requirements file under `examples/chatserver/requirements.txt`
- renamed chatclient.py to test_chatclient.py - for django-nose testrunner
- migrated example project to django 1.7

- edited `docs/testing.rst` to show new changes for using example project

0.4.4

- Added method `release()` to `RedisSubscriber` and calling this method each time a Websocket closes, for whatever reason. This should avoid some reported memory issues.

0.4.3

- Fixed: **django-websocket-redis** failed to initialize under some circumstances in combination with Django-1.7. This only happened for logged in users and threw this exception: `django.core.exceptions.AppRegistryNotReady: Models aren't loaded yet.`
- Added setup on how to run **django-websocket-redis** with uWSGI but without NGiNX.

0.4.2

- Message echoing can be switched "on" and "off" according to the user needs. Before it was "on" by default.
- Many changes to get this app compatible with Python3. This is still not finished, since the pilfered module `websocket.py` is not PY3 compatible yet.
- Added a class `RedisMessage` to pass and store the message to and from the websocket. Before this was just a string with serialized data.

0.4.1

- Fixed: `request.user.username` has been replaced by `get_username()`.

0.4.0

- Messages can be sent to users being member of one or more Django groups.
- `RedisPublisher` and `RedisSubscriber` now only accept lists for `users`, `groups` and `sessions`. This makes the API simpler and more consistent.
- A new magic item `ws4redis.redis_store.SELF` has been added to reflect self referencing in this list, what before was done through `users=True` or `sessions=True`.
- Added the possibility to receive heartbeats. This lets the client disconnect and attempt to reconnect after a number of heartbeats were missed. It prevents silent disconnections.
- Refactored the examples.
- Added reusable JavaScript code for the client.
- Added a context processor to inject some settings from `ws4redis` into templates.

0.3.1

- Keys for entries in Redis datastore can be prefixed by an optional string. This may be required to avoid namespace clashes.

0.3.0

- Added possibility to publish and subscribe for Django Groups, additionally to Users and Sesions.
- To ease the communication between Redis and the Django, a new class `RedisPublisher` has been added as Programming Interface for the Django loop. Before, one had to connect to the Redis datastore directly to send messages to the Websoclet loop.
- Renamed configuration setting `WS4REDIS_STORE` to `WS4REDIS_SUBSCRIBER`.

0.2.3

- Fixed: Use flush to discard received PONG message.

0.2.2

- Moved mokey patching for Redis socket into the runner. This sometimes caused errors when running in development mode.
- Added timeout to select call. This caused IOerrors when running under uWSGI and the websocket was idle.

0.2.1

- Reverted issue #1 and dropped compatibility with Django-1.4 since the response status must use force_str.

0.2.0

- Major API changes.
- Use `WS4REDIS_...` in Django settings.
- Persist messages, allowing server reboots and reconnecting the client.
- Share the file descriptor for Redis for all open connections.
- Allow to override the subscribe/publish engine.

0.1.2

- Fixed: Can use publish to websocket without subscribing.

0.1.1

- Instead of CLI monkey patching, explicitly patch the redis.connection.socket using `gevent.socket`.

0.1.0

- Initial revision.

Credits to Others

When Jacob Kaplan-Moss gave his keynote talk on PyCon 2013 Canada, he mentioned the MeteorJS framework as the next big step in web development.

Personally, I share his opinion about this forecast. The point for both of us is, that we don't see JavaScript as *the* server side language – yet. Probably I am wrong on this, but for the moment I prefer server side frameworks in a language with real classes and numeric types suitable for business applications. This all is missing in JavasSript. Moreover, if content has to be optimized for E-book readers, static rendering on the server side becomes mandatory.

Apart from these technical issues, I love clear separation of concerns, where I can deliberately exchange software components specialized for the running platform. Eventually a web server is very different from a browser, so why should I be forced to run components from the same framework on both of them? If this would be the case, frameworks such as GWT would be more successful.

Therefore my way to go, is for a pure server- and a pure client-side framework. As the latter, I prefer AngularJS, which in my humble opinion is by far the best JavaScript framework ever written.

AngularJS

is a MVC framework for the client with two-way data-binding. Two way data-binding is an automatic way of updating the view whenever the model changes, as well as updating the model whenever the view changes. Django users will immediately feel comfortable with AngularJS, since the concept of templates, controllers and data models is quite similar.

The problem however with two distinct frameworks is, that it becomes difficult to use the server side model on the client, and always keeping track on each model alteration on the server. This by the way, is a typical violation of the DRY principle and should be avoided. I therefore wrote a library, django-angular, which "translates" Django models into an Angular models and vice versa. With this library, for instance, it is possible to use a Django form and bind it with an AngularJS controller without having to keep track on each of the model fields. It is even possible to "export" Django's server side form validation to the client side validation functions, without having to duplicate this code.

Current solutions

For rendering server side data using HTML, and receiving client data through POST or XMLHttpRequests, **django-angular** works fine, but in order to update data on the client upon events triggered by the server, communication using a technology such as websockets must be provided by the application server.

I tried out all of the current implementations to add functionality for websocket to Django. But they all looked like makeshift solutions. Something I found specially disturbing, was the need for another framework running side by side with Django, during development.

uWSGI

Then I stumbled across a talk from Roberto De Ioris on EuroPython 2013.

Here he pointed out, that the WSGI protocol will never be able to support a technology such as websockets. But, since websockets override HTTP, the solution is to let them override WSGI too. Now with a web application runner, supporting thousands of concurrent websocket connections, the implementation for Django was quite easy. Adding a compatible solution for the development environment on Django was somehow trickier, but fortunately Jeffrey Gelens had already implemented a pure Python implementation, which can do the complicated websocket handshake for us.

Since these technologies now can be sticked together, adding three-way data-binding for AngularJS will be the next step. Three-way data-binding is an extension to synchronize changes on the Angular model back to a data-store at the

server side. This is awesome because then Django can manipulate the client side DOM, using the AngularJS template system but without having to implement a single line of JavaScript code. With three-way data-binding, Django will come a step nearer to one of the coolest feature MeteorJS can offer right now.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

G

get_file_descriptor() (ws4redis.subscriber.RedisSubscriber method), 22

P

parse_response() (ws4redis.subscriber.RedisSubscriber method), 22
publish_message() (ws4redis.redis_store.RedisStore method), 22

R

RedisSubscriber (class in ws4redis.subscriber), 22
release() (ws4redis.subscriber.RedisSubscriber method), 22

S

send_persited_messages() (ws4redis.subscriber.RedisSubscriber method), 22
set_pubsub_channels() (ws4redis.subscriber.RedisSubscriber method), 22