latest

Release 0.3.9

Jul 24, 2017

Contents

Created by Stephen McDonald

django-socketio is currently bound to socket.io 0.6, which is considerably out of date. It's fully functional, but some browsers now have newer implentations of WebSockets, and so alternative socket.io transports are fallen back to in these cases.

Work is currently underway to bring django-socketio up to date with the latest gevent-socktio, which has just recently started to support socket.io 0.8

Follow this thread for more info:

https://github.com/stephenmcd/django-socketio/issues/19

CHAPTER 1

Introduction

django-socketio is a BSD licensed Django application that brings together a variety of features that allow you to use WebSockets seamlessly with any Django project.

django-socketio was inspired by Cody Soyland's introductory blog post on using Socket.IO and gevent with Django, and made possible by the work of Jeffrey Gelens' gevent-websocket and gevent-socketio packages.

The features provided by django-socketio are:

- Installation of required packages from PyPI
- A management command for running gevent's pywsgi server with auto-reloading capabilities
- A channel subscription and broadcast system that extends Socket.IO allowing WebSockets and events to be partitioned into separate concerns
- A signals-like event system that abstracts away the various stages of a Socket.IO request
- Support for out-of-band (non-event) broadcasts
- The required views, urlpatterns, templatetags and tests for all the above

CHAPTER 2

Upgrading

Prior to version 0.3, the message argument sent to each of the event handlers was always a Python list, regardless of the data type that was used for sending data. As of 0.3, the message argument matches the data type being sent via JavaScript.

CHAPTER 3

Installation

Note that if you've never installed gevent, you'll first need to install the libevent development library. You may also need the Python development library if not installed. This can be achieved on Debian based sytems with the following commands:

```
$ sudo apt-get install python-dev
$ sudo apt-get install libevent-dev
```

or on OSX using Homebrew (with Xcode installed):

```
$ brew install libevent
$ export CFLAGS=-I/brew/include
```

or on OSX using *macports*:

```
$ sudo port install libevent
$ CFLAGS="-I /opt/local/include -L /opt/local/lib" pip install django-socketio
```

The easiest way to install django-socketio is directly from PyPi using pip by running the following command, which will also attempt to install the dependencies mentioned above:

```
$ pip install -U django-socketio
```

Otherwise you can download django-socketio and install it directly from source:

```
$ python setup.py install
```

Once installed you can then add `django_socketio` to your `INSTALLED_APPS` and `django_socketio.urls` to your url conf:

```
urlpatterns += [
    url("", include('django_socketio.urls')),
]
```

The client-side JavaScripts for Socket.IO and its extensions can then be added to any page with the `socketio` templatetag:

```
<head>
    {% load socketio_tags %}
    {% socketio %}
    <script>
        var socket = new io.Socket();
        socket.connect();
        // etc
    </script>
</head>
```

CHAPTER 4

Running

The `runserver_socketio` management command is provided which will run gevent's pywsgi server which is required for supporting the type of long-running request a WebSocket will use:

```
$ python manage.py runserver_socketio host:port
```

Note that the host and port can also configured by defining the following settings in your project's settings module:

- `SOCKETIO_HOST` - The host to bind the server to.
- `SOCKETIO_PORT` - The numeric port to bind the server to.

These settings are only used when their values are not specified as arguments to the `runserver_socketio` command, which always takes precedence.

Note: On UNIX-like systems, in order for the `flashsocket` transport fallback to work, root privileges (eg by running the above command with `sudo`) are required when running the server. This is due to the Flash Policy Server requiring access to a low port (843). This isn't strictly required for everything to work correctly, as the `flashsocket` transport is only used as one of several fallbacks when WebSockets aren't supported by the browser.

When running the `runserver_socketio` command in production, you'll most likely want to use some form of process manager, like Supervisor or any of the other alternatives.

CHAPTER 5

Channels

The WebSocket implemented by gevent-websocket provides two methods for sending data to other clients, `socket.send` which sends data to the given socket instance, and `socket.broadcast` which sends data to all socket instances other than itself.

A common requirement for WebSocket based applications is to divide communications up into separate channels. For example a chat site may have multiple chat rooms and rather than using `broadcast` which would send a chat message to all chat rooms, each room would need a reference to each of the connected sockets so that `send` can be called on each socket when a new message arrives for that room.

django-socketio extends Socket.IO both on the client and server to provide channels that can be subscribed and broadcast to.

To subscribe to a channel client-side in JavaScript use the `socket.subscribe` method:

```
var socket = new io.Socket();
socket.connect();
socket.on('connect', function() {
    socket.subscribe('my channel');
});
```

Once the socket is subscribed to a channel, you can then broadcast to the channel server-side in Python using the `socket.broadcast_channel` method:

```
socket.broadcast_channel("my message")
```

CHAPTER 6

Broadcast and Send Methods

Each server-side socket instance contains a handful of methods for sending data. As mentioned above, the first two methods are implemented by gevent-socketio:

- `socket.send(message)` - Sends the given message directly to the socket.
- `socket.broadcast(message)` - Sends the given message to all other sockets.

The remaning methods are implemented by django-socketio.

- `socket.broadcast_channel(message, channel=None)` - Sends the given message to all other sockets that are subscribed to the given channel. If no channel is given, all channels that the socket is subscribed to are used. the socket.
- `socket.send_and_broadcast(message)` - Shortcut that sends the message to all sockets, including the sender.
- `socket.send_and_broadcast_channel(message, channel=None)` - Shortcut that sends the message to all sockets for the given channel, including the sender.

The following methods can be imported directly from `django_socketio` for broadcasting and sending out-of-band (eg: not in response to a socket event). These methods map directly to the same methods on a socket instance, and in each case an appropriate connected socket will be chosen to use for sending the message, and the `django_socketio.NoSocket` exception will be raised if no connected sockets exist.

- `django_socketio.broadcast(message)`
- `django_socketio.broadcast_channel(message, channel)`
- `django_socketio.send(session_id, message)`

Note that with the `send` method, the socket is identified by its session ID, accessible via `socket.session.session_id`. This is a WebSocket session ID and should not be confused with a Django session ID which is different.

CHAPTER 7

Events

The `django_socketio.events` module provides a handful of events that can be subscribed to, very much like connecting receiver functions to Django signals. Each of these events are raised throughout the relevant stages of a Socket.IO request. These events represent the main approach for implementing your socket handling logic when using django-socketio.

Events are subscribed to by applying each event as a decorator to your event handler functions:

```
from django_socketio.events import on_message

@on_message
def my_message_handler(request, socket, context, message):
    ...
```

Where should these event handlers live in your Django project? They can go anywhere, so long as they're imported by Django at startup time. To ensure that your event handlers are always loaded, you can put them into a module called `events.py` in one of your apps listed in Django's `INSTALLED_APPS` setting. django-socketio looks for these modules, and will always import them to ensure your event handlers are loaded.

Each event handler takes at least three arguments: the current Django `request`, the Socket.IO `socket` the event occurred for, and a `context`, which is simply a dictionary that can be used to persist variables across all events throughout the life-cycle of a single WebSocket connection.

- `on_connect(request, socket, context)` - occurs once when the WebSocket connection is first established.
- `on_message(request, socket, context, message)` - occurs every time data is sent to the WebSocket. Takes an extra `message` argument which contains the data sent.
- `on_subscribe(request, socket, context, channel)` - occurs when a channel is subscribed to. Takes an extra `channel` argument which contains the channel subscribed to.
- `on_unsubscribe(request, socket, context, channel)` - occurs when a channel is unsubscribed from. Takes an extra `channel` argument which contains the channel unsubscribed from.
- `on_error(request, socket, context, exception)` - occurs when an error is raised. Takes an extra `exception` argument which contains the exception for the error.

- `on_disconnect(request, socket, context)` - occurs once when the WebSocket disconnects.
- `on_finish(request, socket, context)` - occurs once when the Socket.IO request is finished.

Like Django signals, event handlers can be defined anywhere so long as they end up being imported. Consider adding them to their own module that gets imported by your urlconf, or even adding them to your views module since they're conceptually similar to views.

CHAPTER 8

Binding Events to Channels

All events other than the `on_connect` event can also be bound to particular channels by passing a `channel` argument to the event decorator. The channel argument can contain a regular expression pattern used to match again multiple channels of similar function.

For example, suppose you implemented a chat site with multiple rooms. WebSockets would be the basis for users communicating within each chat room, however you may want to use them elsewhere throughout the site for different purposes, perhaps for a real-time admin dashboard. In this case there would be two distinct WebSocket uses, with the chat rooms each requiring their own individual channels.

Suppose each chat room user subscribes to a channel client-side using the room's ID:

```
var socket = new io.Socket();
var roomID = 42;
socket.connect();
socket.on('connect', function() {
    socket.subscribe('room-' + roomID);
});
```

Then server-side the different message handlers are bound to each type of channel:

```
@on_message(channel="dashboard")
def my_dashboard_handler(request, socket, context, message):
    ...

@on_message(channel="^room-")
def my_chat_handler(request, socket, context, message):
    ...
```

CHAPTER 9

Logging

The following setting can be used to configure logging:

- `SOCKETIO_MESSAGE_LOG_FORMAT` - A format string used for logging each message sent via a socket. The string is formatted using interpolation with a dictionary. The dictionary contains all the keys found in Django's `request["META"]`, as well as `TIME` and `MESSAGE` keys which contain the time of the message and the message contents respectively. Set this setting to `None` to disable message logging.

CHAPTER 10

Chat Demo

The "hello world" of WebSocket applications is naturally the chat room. As such django-socketio comes with a demo chat application that provides examples of the different events, channel and broadcasting features available. The demo can be found in the `example_project` directory of the `django_socketio` package. Note that Django 1.3 or higher is required for the demo as it makes use of Django 1.3's `staticfiles` app.