django-omnibus Documentation

Release 0.2.0

Moccu GmbH Co. KG

October 14, 2015

Contents

Django/JavaScript WebSocket Connections.

CHAPTER 1

What is django-omnibus

django-omnibus is a Django library which helps to create websocket-based connections between a browser and a server to deliver messages.

Some use cases could be:

- Chat systems
- Real-time stream updates
- Inter-browser communication
- file transfers
- and so on..

django-omnibus is quite extensible. The connection handler, the Tornado web application and the authenticator can be replaced by just changing the settings. See *Extending django-omnibus* for more detailed information.

For browser compatibility *django-omnibus* also supports SockJS (which provides fallbacks for older browsers).

On the client side, django-omnibus provides a library which handles the connection, authentication and channel subscription (multiple channels can be subscribed using one connection).

CHAPTER 2

Contents

2.1 Server

2.1.1 Installation

Quickstart

To install *django-omnibus* just use your preferred Python package installer:

```
pip install django-omnibus
```

Add `omnibus` to your Django settings

```
INSTALLED_APPS = (
    # other apps
    'omnibus',
)
```

Add the context processor to your Django settings

```
TEMPLATE_CONTEXT_PROCESSORS = (
    # other context processors
    'omnibus.context_processors.omnibus',
)
```

This enables *django-omnibus* with normal websocket support.

Hint: The context processor adds the two variables `OMNIBUS_ENDPOINT` and `OMNIBUS_AUTH_TOKEN` to the template context. You can use these variables to configure the JS library.

Using SockJS

To use SockJS as the underlying transport layer, you have to change some bits.

Install *tornado-sockjs* using your preffered Python package installer:

```
pip install sockjs-tornado
```

Change the following configurations in your Django settings:

```
OMNIBUS_ENDPOINT_SCHEME = 'http'  # 'ws' is used for websocket connections
OMNIBUS_WEBAPP_FACTORY = 'omnibus.factories.sockjs_webapp_factory'
OMNIBUS_CONNECTION_FACTORY = 'omnibus.factories.sockjs_connection_factory'
```

2.1.2 Configuration reference

OMNIBUS_ENDPOINT_SCHEME

This setting is used to construct the connection endpoint for the client connections. You should use `ws` for the default websocket transport layer. If you use SockJS, you have to switch the scheme to `http`, because SockJS starts with a http connection and tries to upgrade to websockets if possible.

OMNIBUS_SERVER_HOST

Used to decide on which address the `omnibusd` server binds. Defaults to all addresses.

OMNIBUS_SERVER_PORT

Sets the port on which the `omnibusd` listens. Defaults to `4242`.

OMNIBUS_SERVER_BASE_URL

The base url of the Tornado handler. If you use the `omnibusd` server, you normally don't have to change this setting. If you want to integrate the omnibus web app in an existing Torndo server, feel free to alter this. Defaults to `/ec`.

OMNIBUS_DIRECTOR_ENABLED

This flag decides if the `omnibusd` process acts as a master node. This means that the process will bind to the `OMNIBUS_PUBLISHER_ADDRESS` instead of connecting. The director is also used as the api endpoint for publishing messages. If you run `omnibusd` with a single instance, you don't have to change this setting. Default is `True`.

To get more information about multi-server setups, please read *Multi-server setups*.

OMNIBUS_FORWARDER_ENABLED

If this flag is changed to `True` (defaults to `False`), the `omnibusd` will start a forwarding proxy for all websocket-clients connecting to the node. The forwarding proxy is used to reduce the amount of connections between `omnibusd` processes. In a single server setup, you don't need this feature.

More details on this topic can be found in the *Multi-server setups* section.

OMNIBUS_SUBSCRIBER_ADDRESS

This is the address, all client connection connect to. Using this connection, the clients receive all their events. Defaults to a local `tcp` address.

If you have a multi-server setup, you can change this address to a remote endpoint. However it is recommended to use the forwarding proxy feature. See `OMNIBUS_FOWARDER_ENABLED`.

OMNIBUS_PUBLISHER_ADDRESS

This address is used to connect to when publishing events. This setting is also used by other python code to publish messages into the *django-omnibus* system. Defaults to a local `tcp` address.

For multi-server setups, you can change this setting to a remote address but you should consider to use the forwarding proxy.

OMNIBUS_DIRECTOR_SUBSCRIBER_ADDRESS

This address is the forwarding destination for local subscriber connections. If a client connects to the local subscriber address, it will receive messages from this upstream server. You need this setting for multi-server setups.

OMNIBUS_DIRECTOR_PUBLISHER_ADDRESS

This address is the forwarding destination for local publisher connections. If a client connection wants to publish and connects to the local publisher address, the forwarding proxy will forward the message to this address. You need this setting for multi-server setups.

OMNIBUS_AUTHENTICATOR_FACTORY

This module path decides which `Authenticator` class is used for authenticating connections. Defaults to `omnibus.factories.noopauthenticator_factory`. This Authenticator does nothing and lets everyone in.

Another valid option is `omnibus.factories.userauthenticator_factory`. This Authenticator identifies Django users including an auth token validation mechanism.

If you want to create your own Authenticator please refer to the existing code to see how it works. The factory is supposed to return a class, not an instance!

OMNIBUS_WEBAPP_FACTORY

This factory returns the Tornado web application which is used by the `omnibsd` server. The shipped options are:

- `omnibus.factories.websocket_webapp_factory` for a websocket app
- `omnibus.factories.sockjs_webapp_factory` for a SockJS app

You can change the factory to extend the way the web application works.

OMNIBUS_CONNECTION_FACTORY

This factory returns the connection class which is used by the web application to handle the connections. One instance is created for every connection. The shipped options are:

- `omnibus.factories.websocket_connection_factory` for a websocket connection
- `omnibus.factories.sockjs_connection_factory` for a SockJS connection

You can change the factory to extend the way the client connection is handled. For example, you could trigger messages when a client connects or disconnects to all other connected clients.

Please refer to the `mousmove` code example and the code itself to see how this works.

2.1.3 Usage (server and API)

Before you can use *django-omnibus*, you have to install the library, please see *Installation* for more details.

Starting the server

Because we use Tornado on the server side to maintain the connections, you have to start the `omnibusd` server in addition to the wsgi application:

```
python manage.py omnibusd
```

In production, you should use `supervisord` or any other process manager to start and stop the omnibus server.

Sending messages to a channel

To send a message to a specific channel, you can use the provided highlevel API.

Let's start with an example:

```
from omnibus.api import publish

def send_hello_world():
    publish(
        'mychannel',  # the name of the channel
        'hello',  # the `type` of the message/event, clients use this name
                  # to register event handlers
        {'text': 'Hello world'},  # payload of the event, needs to be
                                  # a dict which is JSON dumpable.
        sender='server'  # sender id of the event, can be None.
    )

send_hello_world()
```

This would send an message with the type `hello` to the channel `mychannel`. The payload is delivered to all connections which are subscribed to the channel.

A short note about the sender id. Every connection generates an unique id upon connecting. The server-side can decide wether to send an identifier or not and it heavily depends on your application if it is needed or not.

2.1.4 Extending django-omnibus

To extend *django-omnibus* lets have a look at the mousemove example.

In this example project, we subclassed the connection handler and replaced the method `close_connection` to send an event to all other connected clients when the connection is closed.

```
# Example connection.py

from omnibus.factories import websocket_connection_factory


# Our factory function
def mousemove_connection_factory(auth_class, pubsub):
    # Generate a new connection class using the default websocket connection
    # factory (we have to pass an auth class - provided by the server and a
    # pubsub singleton, also provided by the omnibusd server
    class GeneratedConnection(websocket_connection_factory(auth_class, pubsub)):
        def close_connection(self):
            # We subclassed the `close_connection` method to publish a
            # message. Afterwards, we call the parent's method.
            self.pubsub.publish(
                'mousemoves', 'disconnect',
                sender=self.authenticator.get_identifier()
            )
            return super(GeneratedConnection, self).close_connection()

    # Return the generated connection class
    return GeneratedConnection
```

As you can see, we wrote a new connection factory which returns the extended connection handler. This factory is used in the settings like this

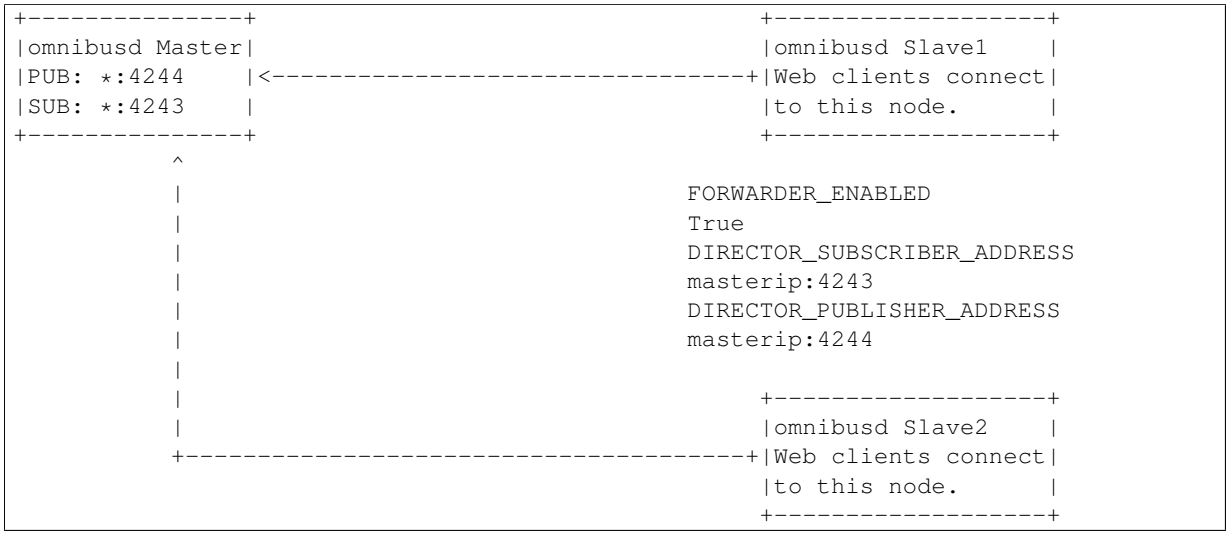
```
OMNIBUS_CONNECTION_FACTORY = 'example_project.connection.mousemove_connection_factory'
```

This is all you have to do to send a "disconnect" event when a client closes the connection.

2.1.5 Multi-server setups

Multi-server setups help you to handle many client connections. Using the forwarding proxy feature, you can have multiple servers accepting omnibus connections.

How it works



This is how you can build a multi server setup.

Assume you have one *master* server which has `OMNIBUS_DIRECTOR_ENABLED` set to `True` and `OMNIBUS_FORWARDER_ENABLED` set to `False`. This server will be your main message hub. Now, you can have one or more *slave* servers, which connect to your *master* server.

On the *slave* servers you then have the `OMNIBUS_FORWARDER_ENABLED` set to `True` and `OMNIBUS_DIRECTOR_ENABLED` set to `False`. Together with the right settings for `OMNIBUS_DIRECTOR_SUBSCRIBER_ADDRESS` and `OMNIBUS_DIRECTOR_PUBLISHER_ADDRESS` the forwarding proxy establishes one subscriber and one publisher connection to your *master* server.

The client connections to the different *slave* servers only connect to local addresses reducing the network IO. This is possible because the forwarding proxy fetches all messages from the *master* and publishes all messages back to the *master* instance.

Hint: Remember to rebind the publisher and subscriber addresses of your *master* servers. The defaults only bind to localhost. Remote servers can't connect to these sockets.

Example configuration with 4 servers

Assuming you have four servers with the ip addresses 192.168.1.10, .11, .12 and .13. This is, how your configuration could look like.

```
# Master server
OMNIBUS_SUBSCRIBER_ADDRESS = 'tcp://192.168.1.10:4243'
OMNIBUS_PUBLISHER_ADDRESS = 'tcp://192.168.1.10:4243'
```

```
# slave servers
OMNIBUS_DIRECTOR_ENABLED = False
OMNIBUS_FORWARDER_ENABLED = True
OMNIBUS_DIRECTOR_SUBSCRIBER_ADDRESS = 'tcp://192.168.1.10:4243'
OMNIBUS_DIRECTOR_PUBLISHER_ADDRESS = 'tcp://192.168.1.10:4243'
```

You now could define a DNS round-robin entry like "omnibus.myfancydomain.com" pointing to 192.168.1.11, 192.168.1.12 and 192.168.1.13 to create some kind of a poor man's load balancing.

2.1.6 Internals / behind the scenes

This document describes some of the internals of *django-omnibus*.

The libary uses Tornado and ZMQ for handling the connections and message delivery.

Message envelop and structure

Let's have a look on a full event blob

```
mychannel:{"type":"foobar","sender":"44b0b696-8048-4ac3-9219-ca8d81a26879",
"payload":{"arg1":123,"example":"string"}}
```

The deconstructed envelope:

```
mychannel  // the channel name
:  // a colon to split channel and message.
{"type":"foobar","sender":"44b0b696-8048-4ac3-9219-ca8d81a26879",
"payload":{"arg1":123,"example":"string"}}  // the json message
```

In this message, `mychannel` is the destination channel of the message. The colon divides the destination and the message.

The deconstructed event body:

- `type` which is used to trigger the right event handlers on the client side.
- `sender` identifies the origin of the event
- `payload` can be anything from simple key-value pairs to large JSON blobs.

Commands

There is a special case if the channel starts with a bang `!`. All messages which start with a bang are considered commands from the client to the server and are never forwarded.

An example, this is used to subscribe to a channel:

```
!subscribe:mychannel
```

Here is the response

```
!subscribe:{"type":"subscribe","success":true,"payload":{"channel":"mychannel"}}
```

The response to a command looks the same as normal messages with some notes:

- the destination channel is `!` + command name, e.g. `subscribe`
- the message type is the command name, e.g. `subscribe`
- there is an additional field `success` indicating whether the command was successful or not.

The payload can contain various things. In the case of subscribe it contains the channel which was subscribed.

2.2 Client

2.2.1 Installation and integration

Quickstart

The JavaScript client implementation is shipped with the django module. After you've installed the `django-omnibus` according the documentation you are able to insert the script as followed

```
{% load static %}
<script type="text/javascript" src="{% static 'omnibus/omnibus.min.js' %}"></script>
```

All JavaScript sources are available as non-minified or minified version. The minified scripts uses `.min.js` as extension.

`django-omnibus` gives you the possibility to use them as an AMD module to use with RequireJS or as CommonJS-like environments that support `module.exports` such as browserify.

Dependencies

The client library is *dependency free*. When you want to support older Browsers which don't support WebSockets and/or JSON by default, embed the following libraries. These are also shipped with the *django-omnibus* module:

SockJS

SockJS can be used as alternative transport implementation. It delivers the same API as the browsers standard Web Websockets.

```
<script type="text/javascript" src="{% static 'omnibus/sockjs.min.js' %}"></script>
```

or get the latest version from SockJS via GitHub

JSON2

JSON2 implements the functions `JSON.parse` and `JSON.stringify` for browsers without the standard JSON API.

```
<script type="text/javascript" src="{% static 'omnibus/json2.min.js' %}"></script>
```

or get the latest version from JSON2 via GitHub

Setup

After inserting the `django-omnibus` JavaScript library as described above, you can follow these steps to setup a connection

```
// Select a transport implementation:
var transport = WebSocket; // SockJS can be used alternatively

// Receive the path for the connection from the django template context:
var endpoint = '{{ OMNIBUS_ENDPOINT }}';

// Define connection options:
var options = {
    // Get the omnibus authentication token:
    authToken: '{{ OMNIBUS_AUTH_TOKEN }}'
};

// Create a new connection using transport, endpoint and options
var connection = new Omnibus(transport, endpoint, options);
```

After you've created an instance, it will automatically open a connection and identify itself through them. For more informations about the connection instance visit the *usage* site.

The communication between client and server takes place through a channel. You can easily open a channel instance and send a *ping* using the following lines of code

```
var channel = connection.openChannel('ping-channel');
channel.send('ping-event', {
    message: 'hello world'
});
```

For more information about the use of a channel instance take a look at the *usage* site or visit the examples.

2.2.2 Usage

Connection

A connection can simply be created using the `Omnibus` constructor. To open an instance provide as the constructors parameters:

- `transport`, it describes the implementation of the web socket transport medium. This can be the browser's default implementation `WebSocket` or alternatively `SockJS`.
- `endpoint`, is the location where the remote listens.
- `options`, can optionally specify the behaviour of the connection.

```
var transport = WebSocket;
var endpoint = 'ws://localhost:4242/ec';
var connection = new Omnibus(transport, endpoint);
```

Options

In the code above are no options provided, so the instance use their default options. To take a look at the default options you can access the `Omnibus.defaults` property. The following options can be provided as property:

- `authToken`, a string which identifies a client connection. The string will be generated by the remote.
- `ignoreSender`, a boolean (default `true`) defines if you send a message through a channel and the remote just forwards the message to their receivers, it ignores your own message and doesn't triggers any action.
- `autoReconnect`, a boolean (default `true`) enables an auto-reconnect when the connection to the remote gets lost unexpectedly.
- `autoReconnectTimeout`, a number (default `500`) is only used when `autoReconnect` option is enabled. It describes the timeout in milliseconds when the next try to connect to the remote will be performed.

```
var transport = WebSocket;
var endpoint = 'ws://localhost:4242/ec';
var options = {
    ignoreSender: true,
    autoReconnect: true,
    autoReconnectTimeout: 1000
};
var connection = new Omnibus(transport, endpoint, options);
```

Public functions

- `getId`, returns the unique identifier of this connection which is used to communicate with the remote.
- `isAuthenticated()`, returns whether the connection was authenticated at the remote.
- `isConnected()`, returns if the connection is established.
- `openChannel(name)`, returns a channel with the given name. When a channel with the same name was opend previously, then it returns the same channel instance as before. Otherwise it instantiates a new channel with the given name. When a previously returned channel instance is already closed, a new instance will be generated. For more information about a channel instance take a look into the *channel* section.

- `getChannel(name)`, returns a channel instance which was opend previously through this connection. If there was not opened a channel with the given name before the function returns 'undefined'.
- `closeChannel(instanceOrName)`, closes and finally destroys a channel which was opened through this connenction.

The connection has some more public functions. Some of them are not intended to be used directly. The others are to be used with the omibus eventbus. The connection fires some predefined events. For more details according events see the *events* section below.

Channel

On an open connection its easy to open a channel. The channel must exist at the remote. For example you would like to send or receive messages from the channel *foo*, the subscription would look like this:

```
var foo = connection.openChannel('foo');
```

To send a message through this channel you call:

```
// send message with type 'bar' through channel:
foo.send('bar');

// send mesage with type 'baz' and data through channel:
foo.send('baz', {
    omg: 'wtf'
});
```

To receive messages from the remote through an open channel add an eventhandler to handle the message and perform actions:

```
foo.on('bar', function(event) {
    // perform some custom action here...
});
```

The channel fires some predefined events. For more details according events see the *events* section below.

To close a channel instance call the `close()` function. This triggers an unsubscription from the connection. Finally it closes the channel from the remote and calls `destroy()` indirectly.

The `destroy()` function is not meant to be called directly. This will be called through the instance or the referred connection. When called all registered *events* will be removed from the called instance.

Events

The *django-omnibus* ships his own eventbus system. The *connection* and *channel* classes inherit from the eventbus. There are some functions to interact with the eventbus sytem:

- `on(eventName, eventHandler)`, registers an eventHandler to a particular eventName. This function is chainable to call multiple functions on an instance of the eventbus. To handle all events triggered through an eventbus instance by a single eventHandler use the value `'*'` as wildcard eventName.
- `off(eventName, eventHandler)`, removes registered eventHandler(s) from the eventbus instance. This function is chainable to call multiple functions on an instance of the eventbus. When called without any parameters all registered eventHandlers are removed. When called with eventName parameter all registered eventHandlers according this particular eventName are removed. When called with eventName and a reference to an eventHandler function this given handler for the particular eventName will be removed.
- `trigger(eventName, eventData)`, triggers all registered eventHandlers on a particular eventName. EventData can be send through each call to each handler. This function is chainable to call multiple functions on an instance of the eventbus. All eventHandlers registered with the wildcard eventName `'*'` will be triggered as well.

All predefined event types are listet through the `Omnibus.events` property. The containing constants are prefixed with `CONNECTION` or `CHANNEL` to specifiy the instance who will fire this event.

- `Omnibus.events.CHANNEL_SUBSCRIBED`, notifies about a channel subscription state.
- `Omnibus.events.CHANNEL_UNSUBSCRIBED`, notifies about the current channel unsubscription state.
- `Omnibus.events.CHANNEL_CLOSE`. notifies that the channel instance will be closed.
- `Omnibus.events.CHANNEL_DESTROY`, notifies that the channel instance will be destroyed and isn't available for further usage.
- `Omnibus.events.CONNECTION_CONNECTED`, notifies about an established connenction.
- `Omnibus.events.CONNECTION_DISCONNECTED`, notifies about a (may be accidentally) closed connection.
- `Omnibus.events.CONNECTION_AUTHENTICATED`, notifies about a successful identification.
- `Omnibus.events.CONNECTION_ERROR`, notifies about an occurred error through the websocket implementation.

An example usage looks like this:

```
var transport = WebSocket;
var endpoint = 'ws://localhost:4242/ec';
var connection = new Omnibus(transport, endpoint);

connection
    .on(Omnibus.events.CHANNEL_SUBSCRIBED, function(event) {
        // handle subscription actions here...
    })
    .on(Omnibus.events.CONNECTION_AUTHENTICATED, function(event) {
        // handle authentication actions here...
    });
```

2.2.3 Internals / behind the scenes

All sources of the JavaScript client library are located at `omnibus/static/omnibus/src/`. If you want to change or add any feature be sure to edit only those JavaScript files. The files are organized as *AMD Modules* and will be packed into a single file using the build process.

Utils

utils are located inside the source folder. Each file is a simple function to help simplify the rest of the JavaScript code.

The `extend()` function is a simpler reimplementation of the jQuery extend() function. It's used to simplify the prototypal inheritance. For example:

```
var Mother = function() {};
var Child = function() {};

// Child inherts Mother and gets some extra properties 'foo' and 'bar':
extend(Child.prototype, Mother.prototype, {
    foo: 'baz',
    bar: function() {
        return this.foo();
    }
});
```

The `proxy()` function is another reimplementation of the jQuery proxy() function. It's described as a function which *"takes a function and returns a new one that will always have a particular context"*. It's used to keep the current instance context and not to use such code like `var self = this;`.

```
extend(Child.prototype, Mother.prototype, {
    baz: 'foo bar baz',
    foo: function() { //without proxy()
        var self = this;
        window.setTimeout(function() {
            window.alert(self.baz);
        }, 1000);
    },
    bar; function() {
        window.setTimeout(proxy(
            function() {
                window.alert(this.baz);
            },
            this
        ), 1000);
    }
});
```

EventBus

The EventBus implements the pubsub/observer pattern. The *Connection* and *Channel* instances inherits from the EventBus. To add and remove handlers for a certain *event name* use the `on()` and `off()` functions. To notify those handlers the `trigger()` function is used.

Registration

All registered handlers are stored in an Array property of the `_events` property. To access all registered handlers for a certain `eventName` can be done by `this._events[eventName]`.

Wildcard handlers

To add a function which will be executed for all triggered events use the `'*'` String.

Event instances

The executed functions are parametrized by an instance of the Event Class. The instance contains 3 properties `name`, `data` and `sender`:

- `name` is the event name which was triggered
- `data` contains *optinal* extra data which is transported through the event.
- `sender` is the reference to the instance which triggered the event.

Connection

Channel reference storage

All references to previously opened channels using `openChannel()` are stored using the dictionary object `_channels`. Each channel can be accessed by their `channelName` via `this._channels[channelName]`. Each name/channel combination is handled as single instance. There won't be two different instances of the same channel through the same connection. Only when a channel was closed before, the reference form the `_channels` dictionary is removed also.

The send queue

The send queue allows the user of the JavaScript library simply to start sending messages without taking care whether the connection is already established and authenticated or not. They are able to open channels and send messages and don't have to wait for events which allow further activities.

The send queue is represented by the `_sendQueue` property of a connection instance. Its a simple Array which stores the messages which wasn't send, because the connection is not connected to the remote or authenticated. When both conditions are complied, the `_flushQueue()` function synchronously sends the queued messages to the remote.

Channel

Closing a Channel

To close a channel can be achieved in different ways. It can be called through the channel instance using `channel.close()` or through the connection where the channel was created with `connection.closeChannel(channel)`.

When a channel is closed, its instance will be destroyed afterwards.

1. When calling `channel.close()` the channel triggers the *"close"* event
2. An unsubscribtion command will be send to the remote by the `closeChannel()` function
3. The remote will answer
4. The channel will fire the *"unsubscribed"* event
5. The channel will be removed from the connection
6. The channel will be detroyed and fires the *"destroy"* event

GruntJS Taskrunner

Configurations

The GruntJS task configurations are not as usual stored in the `Gruntfile.js`. They are located in `resources/grunt-configs/`. Each task is represented in a single file with the tasks name. The *grunt-'prefix is not mentioned in the filename for the reason that all tasks have the same prefix. For example the configuration for the '"grunt-contrib-jshint"*-Task is located in `resources/grunt-configs/contrib-jshint.js`.

Build a new release

To build a release we pack all *AMD Modules* into a single file. For this workflow we use the grunt-contrib-requirejs task and hook into it, using the `onBuildWrite` and `onModuleBundleComplete` callbacks.

Inside the `onBuildWrite` we remove the typical `define()` and `return` statements which defines each single module. For example this sample `Foo` module looks without the `onBuildWrite` callback like this:

```
define('./Foo', [], function() {
    var Foo = function() {};
    Foo.prototype.bar = function() {
        window.alert('baz');
    }

    return Foo;
});
```

The callback removes the statements which results in:

```
var Foo = function() {};
Foo.prototype.bar = function() {
    window.alert('baz');
};
```

In the `onModuleBundleComplete` we wrap some additonal code around our output. To wrap the code we use a template located in `/omnibus/static/omnibus/src/wrapper/wrapper.js.tpl`. It contains the code to allow the library to use as *AMD Module*, as *CommonJS Module* or with plain JavaScript.

This approach has some pitfalls. When using for example module `Foo` in another module `Bar`, then **it's important to use the same module names in both files**. This example should show why:

Without the `onBuildWrite` task the output looks like this:

```
// The result from Foo.js:
define('./Foo', [], function() {
    var Foo = function() {};
    Foo.prototype.bar = function() {
        window.alert('baz');
    };

    return Foo;
});

// The result from Bar.js:
define('./Bar', ['./Foo'], function(Foo) {
    var Bar = function() {};
    Bar.prototype.go = function() {
        return new Foo();
    };

    return Bar;
});
```

When removing the wrapped statements it results in:

```
// The result from Foo.js:
var Foo = function() {};
Foo.prototype.bar = function() {
    window.alert('baz');
};

// The result from Bar.js:
var Bar = function() {};
Bar.prototype.go = function() {
    return new Foo();
}
```

As you can see, the `Bar` module needs the same name for the `Foo` reference as the `Foo` module uses itself.

2.3 Contribution

If you like to contribute to this project please read the following guides and read the internals sections for *Server* and *Client* code.

2.3.1 Django Code

To install all requirements for development and testing, you can use the provided requirements file.

```
$ pip install -e .[tests]
```

Testing the code

django-omnibus uses `py.test` for testing. Please ensure that all tests pass before you submit a pull request. `py.test` also runs PEP8 and PyFlakes checks on every run.

This is how you execute the tests and checks from the repository root directory.

```
$ py.test
```

If you want to generate a coverage report, you can use the following command.

```
$ py.test --cov=omnibus --cov-report=html .
```

Documentation

django-omnibus uses Sphinx for documentation. You find all the sources files in the `docs/source` folder.

To update/generate the html output of the documentation, use the following command inside the `docs` folder.

```
$ make html
```

Please make sure that you don't commit the build files inside `docs/build`.

2.3.2 JavaScript Code

The client side development depends on the GruntJS Taskrunner. If you haven't used grunt before, be sure to check out the Getting Started guide.

To setup your local environment call `npm install` in the projects root. This downloads all necessary dependencies to run the taskrunner.

Change the code

A build of the JavaScript library will be made on each new release.

The release is based on the *AMD Modules* located in the source directory *"src"* inside the *"static"* folder. For these circumstances it's not meant to made changes in the JavaScript files outside the *"src"* folder, cause they are overwritten by the build output. If you wan't to contribute to this project please commit only the *src*-files.

To run the build process you can call the grunt taskrunner using:

```
$ grunt
```

This starts *validation*, *testing*, *building* and *documention* processes in a row as default task of the taskrunner. The build process creates the library itself and a minified version of them, using the extention `.min.js`.

Generate a documentation

The client code is fully documented using JSDoc. To get an overview about the classes and functions generate your own JSDoc running the following command. The generated documentation will open immediately.

```
$ grunt doc
```

Validation & testing

Before you commit your code changes and offer a pull request run the following tasks via grunt:

To validate the code according our JSHint, JSCS and indentation rules run:

```
$ grunt validate
```

To finally test your JavaScript code run:

```
$ grunt test
```

The tests are written using Jasmine. The test specs are located at `testing/jstests/`.

Code declaration

The most JavaScript code will be validated as described above using JSHint and JSCS. But there are some rules which won't be checked:

- To declare a private property, add a leading underscore _ to the properties name, for example: `_isValid: true`.
- To declare a constant, use uppercase letters and underscores like: `THIS_IS_A_CONSTANT = 'value'`.

CHAPTER 3

Indices and tables

- genindex
- search