telegraphy Documentation

Release 0.1

Carlos de la Torre, Nahuel Defossé

December 13, 2013

Contents

Project home: https://github.com/machinalis/telegraphy/

The Telegraphy Project provides real time events for WSGI Python applications with additional features such as event filtering, subscription persistence and authorization/authentication.

It's initially intended for Django but you can extend it to any WSGI framework.

WebSocket pub/sub and RPC is based on AutobahnPython implementation of WAMP protocol

CHAPTER 1

Getting It

You can get *Telegraphy* by using pip:

```
$ pip install telegraphy
```

Or grab the source code from the GitHub repository and run setup.py:

```
$ git clone git://github.com/machinalis/telegraphy/telegraphy.git
$ cd telegraphy
$ python setup.py install
```

For more detailed instructions check out our *Installation instructions*. Enjoy.

CHAPTER 2

Get Involved

We are eager to hear from the community: to receive suggestions, bug-reports, participate in discussions and improve Telegraphy as much as it's possible.

For all of that, we have a Google group in http://groups.google.com/group/telegraphy

Also, to guide the development efforts and issues, we are using GitHub's issue tracker.

CHAPTER 3

Contents

3.1 Installation instructions

3.1.1 Get It

Pip

You can get *Telegraphy* by using pip:

```
$ pip install telegraphy
```

You will need to have pip installed on your system. On linux install the python-pip package, on windows follow this. Also, if you are on linux and not working with a virtualenv, remember to use `sudo` for both commands (`sudo pip install telegraphy`).

Download

Download the latest packaged version from http://pypi.python.org/pypi/telegraphy/ and unpack it. Inside is a script called setup.py. Enter this command:

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

...and the package will install automatically.

Source code

Telegraphy is hosted on github:

```
https://github.com/machinalis/telegraphy
```

Source code can be accessed by performing a Git clone.

The following command will check the application's source code out to a directory called *telegraphy*:

Git:

```
$ git clone git://github.com/machinalis/telegraphy/telegraphy.git
```

You should either install the resulting project with *python setup.py install* or put the *telegraphy* directory in your PYTHONPATH.

You can verify that the application is available on your PYTHONPATH by opening a Python interpreter and entering the following commands:

```
>>> import telegraphy
```

No exceptions should raise.

Keep in mind that the current code in the git repository may be different from the packaged release. It may contain bugs and backwards-incompatible changes but most likely also new goodies to play with.

System dependencies

One of our main dependencies, *Twisted*, requires gcc which is not available by default.

In Ubuntu-like systems you'll need to install `python-dev`:

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

To build the documentation, you'll need to have Sphinx installed:

```
$ pip install Sphinx
```

To help in the documentation elaboration, we have a small script that detects changes while you are working and automatically builds the doc: `./autobuild-docs.sh` . If you want to use it, you'll need *inotify*:

```
$ sudo apt-get install inotify-tools
```

3.1.2 Installing the Django app

Telegraphy's Django app is bundled within the `contrib` directory in the Telegraphy root.

It is installed with the standard procedure: in your project's *settings.py* file add `telegraphy.contrib.django_telegraphy` to the `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    ...
    'telegraphy.contrib.django_telegraphy',
    ...
)
```

The default TEMPLATE_CONTEXT_PROCESSORS do not include the request as a variable in the context so, if you haven't done so yet, add `django.core.context_processors.request`:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.request',
    ...
)
```

3.2 The Telegraphy Project

This project is about making the *real-time web* easier for the Django developers (and to Python web-developers in general).

In a very general way, the issue we want to solve is **How can I easily receive and handle server-side generated events, on the frontend?**.

There's a lot going on about this. There are many standards, protocols, tools, services and sophisticated frameworks related to this issue. But most of them have at least one of the following problems:

- They don't solve the whole problem
- They are not easy to use.
- They are not well documented.

Therefore, our main objectives are to provide:

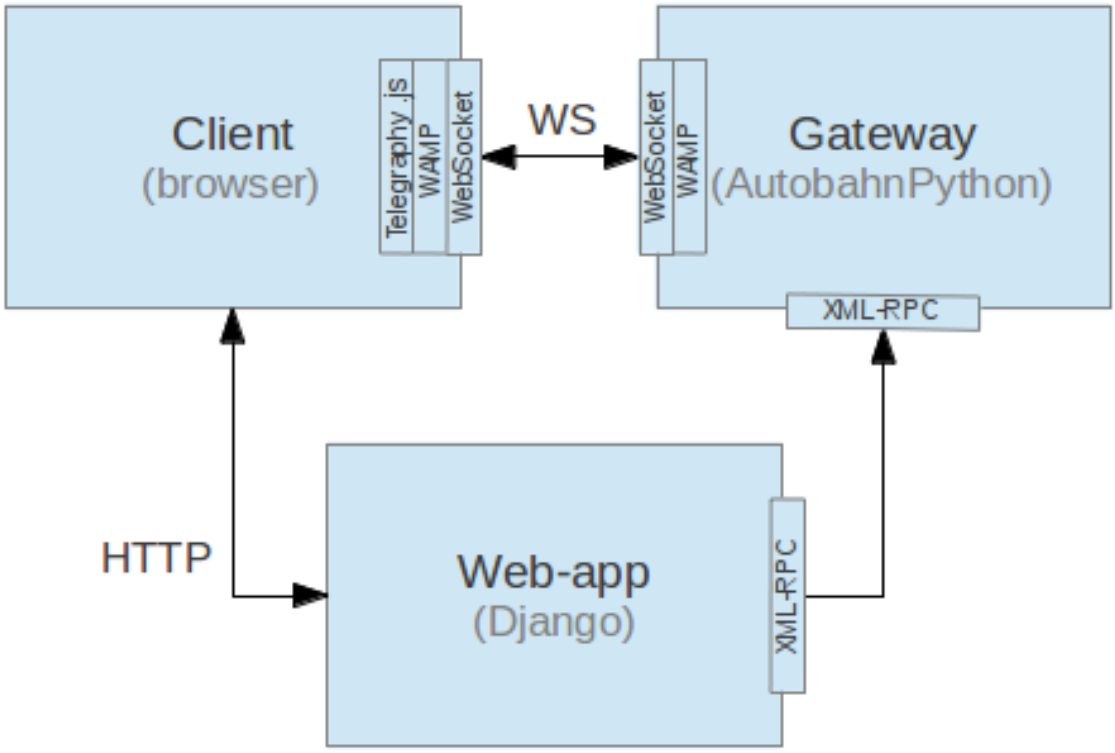
- well documented and tested tools,
- that are simple to install and use,
- which rely on open standards and protocols,
- to emmit and handle asynchronous, server-side events in real-time.

3.2.1 Architecture

Overview

The Telegraphy project's architecture has three main components:

- A **web-application** that registers and emits events.
- A **gateway** is a scalable, high-performance, asynchronous, networking engine.
- A **client** api which talks WAMP (through a WebSocket) with a *Gateway*. This is normally a JavaScript loaded from a web-page.



One of the objectives of the project is to keep these main components decoupled. For each one of them there are several available technologies (and more will appear). For example:

- The web-app can be anything from a full-scale desktop or web application to a simple script.
- **The gateway can be implemented using Twisted, Tornado,** Node, ...
- The web-app and the gateway can communicate through an XML-RPC lib, or using message queues such as ØMQ, RabbitMQ, ...
- The client can be anything implementing *WAMP* over WebSockets, typically a Javascript program.

The current implementation is based on Django for the web-app and client side components. The gateway is implemented using Twisted.

Django app A very useful app is provided. It features a class-based mechanism to extend the application's models with the capability to generate server-side events.

Also, template tags and a Javascript API (based on AutobahnJS) are provided. These make it really easy to handle the events on the client side.

Gateway Currently, a Twisted-based server using AutobahnPython).

The web-app and gateway communicate through XML-RPC with a shared-configuration approach.

3.3 Gateway

WORK IN PROGRESS!!!

3.3.1 Interface with the client

The general idea behind the **gateway** is to provide an asynchronous-events management server.

Its interface with the client applications is based on the WAMP protocol. The current version relies on Autobahn-Python, which provides a fully asynchronous Twisted-based implementation.

3.3.2 Interface with the web-app

On the other side, the interface with the web-app is not yet fully defined. Currently, the *gateway* receives event's data through XML-RPC, which is very general and flexible, but not highly performant.

As the project's functionalities develop and new features are added, the comunication of the web-app and the gateway will be encapsulated in a specific API, whose implementation will probably use some message-passing tool, such as ZMQ, RabbitMQ, etc.

The current ideas are based in some *proxy* module that provides a uniform interface for the web-app. Such proxy communicates with a gateway *representative*, that is connected-to/part-of the current running gateway instance. The *proxy* and *representative* can be in sync by a shared-configuration policy.

Such architecture would allow the web-app to be independent of the communication mechanism with the gateway. Many implementations can be maintained and used as needed (XML-RPC, message-queue, etc.).

3.3.3 Features (Notes)

Provides an asynchronous-events management server.

Web-app agnostic: even when the current implementation uses Django, the design of the gateway shall be completely independent of the web-app technology. Ideally, the interfaces should be open, standard and well defined so that any program should be able to interact with the gateway (even non-Python programs!).

The gateway has the responsability to assure continuous service. Changes in configuration or events definitions must be transparent for the client (if possible). Otherwise, specific resources must be design in order to be able to implement client-side mechanisms to remain "connected" (reconnect, etc).

Client representatives identification: on connection, the Gateway provides a unique identifier (token). The representatives saves the token in a cookie. The cookie has an expiration time defined by the Gateway.

Persistent subscriptions: a client may decide that a given subscription to an event is 'permanent'. The subscription mechanism provided by the protocol must include some parameter to indicate this situation.

- **Real Time Events**
  - Authentication
  - **Subscription handling**
    - Public vs Authenticated Events
    - Subscription management (client or event based)
  - **Event management**
    - Class based event definition
    - **Event query language**
      - Performance
      - Simplified client side subscription handling
      - Easy channel emulation

3.4 django-telegraphy

Telegraphy aims to facilitate the integration of real-time features into a Django project.

Django is not yet prepared for handling real time web features. There are a lot of issues and technologies that must be taken into account that are not trivial to integrate with Django: WebSockets, asynchronous servers, message queues, advanced key-value stores, etc.

Telegraphy takes care of all that. It provides a simple, class-based way to provide your models with the capability to generate events. These will reach the client application, in near-real-time. Besides:

- A management command to run the *Gateway*.
- Automatic signals-based CUD events (Create, Update, Delete).
- Custom events definitions.
- Template tags for easy configuration.
- A very simple JS API to make real-time Django apps a reality.

For installation notes, refer to the *Installation instructions*.

3.4.1 Management command: run_telegraph

django-telegraphy provides a management command to run the *Gateway* server. Until this functionality can be properly integrated in the existing `runserver` command, open a new console and run:

```
$ python manage.py run_telegraph
```

For more info about this process, go to the *Gateway* section.

3.4.2 Class based events generation

Currently, this app's main feature is to give your models the ability to generate server-side events that will reach the clients. To do that you must:

- Create an `events.py` file in your app's directory (next to the `models.py` and `urls.py`).
- For every model you want to generate events, create a sub-class of `telegraphy.contrib.django_telegraphy.events.BaseEventModel`:

```
from models import MyModel
from telegraphy.contrib.django_telegraphy.events import BaseEventModel


class MyEventsModel(BaseEventModel):
    model = MyModel
```

- Register the event:

```
event = MyEventsModel()
event.register()
```

The `events` module provides an `autodiscover` method to automatically register all the events in the app. This method is typically called in the project's `urls.py` file:

```
from django.conf.urls import patterns
...
from telegraphy.contrib.django_telegraphy import events


events.autodiscover()

urlpatterns = ...
```

- Done. Every time an instance of `MyModel` is created, saved or deleted, an event will be generated and sent to the clients through the *Gateway*.

BaseEventModel

`telegraphy.contrib.django_telegraphy.events.BaseEventModel`. The class' allowed attributes are:

model It is the only compulsory attribute. It references the target Django.db Model being extended.

fields (*default = None*) A list of the target model's field names to include in the event data. If it is `None` then all the Model's fields will be included.

If the target model has a `to_dict()` attribute, then it is used to generate the event's data, so the `fields` attribute is ignored.

If this attribute is set, then exclude is ignored.

exclude (*default = None*) A list of the target model's field names to ignore. They will not be sent in the event data, in the case that `fields` is `None`.

If the target model has a `to_dict()` attribute, then it is used to generate the event's data, so the `exclude` attribute is ignored.

operations (*default = (OP_CREATE, OP_UPDATE, OP_DELETE)*) Indicates what operations in the target model's instances will generate events:

`OP_CREATE`: an event will be generated each time an instance of the target model is **created** .

`OP_UPDATE`: an event will be generated each time an instance of the target model is **saved** (not on creation).

`OP_DELETE`: an event will be generated each time an instance of the target model is **deleted**.

If *false-ish* then no events will be automatically generated by this model.

name (*default = None*) The name of the event. If none is provided, it is automatically generated from the app and target model's name/ For example: *myapp.MyModel*.

verbose_name (*default = None*) A verbose, human-friendly name for the event. If provided, it is sent in the event's meta-data. A helper for user-interface purposes.

Autodiscover

The `events` module provides an `autodiscover` method. It will search for all the models that are subclasses of BaseEventModel, in all the project's installed apps. Then it instantiates and registers each one of them.

3.4.3 Events

TBD.

3.4.4 Authorization

TBD.

3.4.5 Communications with the gateway

XML-RPC based. TBC.

3.4.6 Generating custom events

TBD.

3.4.7 Template tags

TBD.

3.4.8 JavaScript API

TBC. The AutobahnJS-based API provides a Gateway *representative* which is responsible of:

- connect to a running instance of a Gateway
- subscribe to events. Free events? can we subscribe to unregistered Gateway events?
- provide means to handle connection changes (keep the connection alive?)

3.5 Authentication shortcomings

Django uses a **HTTP Only** cookie called *sessionid*. This cookie would not be exposed to JavaScript for security issues. Since Gateway process may not run in the same context (port, ip, machine) where Django is running, we can't rely on it for authentication.

In order to authenticate clients we must pre share (key, secret) tokens, as part of the WAMP Challenge-Response-Authentication mechanism. This tokens are created by the gateway whenever a page that uses telegraphy's template tag is rendered. These tokens are short lived, they expire once the websocket connection has been established.

TBC.

3.6 Examples

3.6.1 The basics

1. Run the *Gateway* server:

   ```
   $ python manage.py run_telegraph
   ```

2. Create an `events.py` file in your app's directory (next to the `models.py` and `urls.py`).
3. For every model you want to generate events, create a sub-class of `telegraphy.contrib.django_telegraphy.events.BaseEventModel`:

```
from models import MyModel
from telegraphy.contrib.django_telegraphy.events import BaseEventModel


class MyEventsModel(BaseEventModel):
    model = MyModel
```

4. Register the event:

```
event = MyEventsModel()
event.register()
```

The `events` module provides an `autodiscover` method to automatically register all the events in the app. This method is typically called in the project's `urls.py` file:

```
from django.conf.urls import patterns
...
from telegraphy.contrib.django_telegraphy import events


events.autodiscover()

urlpatterns = ...
```

5. Create you template, including *Telegraphy template-tags*

```
{% load telegraphy_tags %}
{% load static %}
<html>
    <head>
        <title>Simple Telegraphy API Example</title>
        <script src='{% static "telegraphy_demo/js/jquery-1.10.2.js" %}'></script>
        {% telegraphy_scripts %}
    </head>
    <body>
        <h1>Catching model events!</h1>
        <ul id="event_catcher"> </ul>
        <script>
            (function (){
                var $event_catcher = $('#event_catcher');
                Telegraphy.register('telegraphy_demo.MyModel',
                    function (tEvent){
                        console.log("Event", tEvent);
                        var new_line = $('<li/>').text("New instance");
                        $event_catcher.append(new_line);
                    });
            })();
        </script>
    </body>
</html>
```

3.6.2 More examples

The demo_project in the repo includes a simple example page, very similar to the shown above.

Another more feature-rich, yet still simple example, is included in the change tracker page. In this case, the models and events are the same, only the HTML and JS code changes.

3.7 WAMP Authentication and Extended Session

A gateway is published in a URL. Many gateway instances can coexist. Telgraphy uses WAMP RPC/PubSub capabilities adding some authentication mechanism.

3.7.1 Server RPC Methods

- **/get_token**

  Params: None

  Result: token (string)

  Call made by the web application to the gateway to generate a unique valid authorization token to be given to browser for later call to authenticate by the client.

- **/authenticate**

  Params: auth_token (string), previous_session (string, optional)

  Result:

  - In any case when auth_token is invalid, CALLERROR will be returned.
  - If pervious_session is empty or null, CALLRESULT idicates success.
  - **If previous_session is not null**
    - If previous_session matches to an existing gateway client, CALLRESULT indicates success.
    - Otherwise CALLERROR will be returned.

  Call from the client(browser) to the gateway to authenticate.

  If it's new connection, only auth_token will be present. If the client is reconnecting, the previous_session will be sent.

- **/subscriptions** (optional)

  Params: None Result: List of URIs o CURIs subscribed by the client.

- **/subscriptors** (optional)

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*