Flask-SSE Documentation

Release 1.0.0

David Baumgold

Jul 11, 2021

Contents

1 Quickstart **3**

2 Configuration **5**
- 2.1 Redis . . . 5
- 2.2 Application Server . . . 5

3 Advanced Usage **7**
- 3.1 Channels . . . 7
- 3.2 Access Control . . . 7

4 API **9**

5 Indices and tables **11**

Index **13**

A Flask extension for HTML5 server-sent events support, powered by Redis.

CHAPTER 1

Quickstart

Here's a quick working example of how Flask-SSE works.

Warning: Server-sent events do *not* work with Flask's built-in development server, because it handles HTTP requests one at a time. The SSE stream is intended to be an infinite stream of events, so it will never complete. If you try to run this code on with the built-in development server, the server will be unable to take any other requests once you connect to this stream. Instead, you must use a web server with asychronous workers. Gunicorn can work with gevent to use asychronous workers: see gunicorn's design documentation.

You will also need a Redis server running locally for this example to work.

Make a virtual environment and install Flask-SSE, `gunicorn`, and `gevent`. You will also need to make sure you have a Redis server running locally.

```
$ pyvenv sse
$ pip install flask-sse gunicorn gevent
```

Make a file on your computer called `sse.py`, with the following content:

```
from flask import Flask, render_template
from flask_sse import sse

app = Flask(__name__)
app.config["REDIS_URL"] = "redis://localhost"
app.register_blueprint(sse, url_prefix='/stream')

@app.route('/')
def index():
    return render_template("index.html")

@app.route('/hello')
def publish_hello():
    sse.publish({"message": "Hello!"}, type='greeting')
    return "Message sent!"
```

If you are using a Redis server that has a password use:

```
app.config["REDIS_URL"] = "redis://:password@localhost"
```

Make a `templates` folder next to `sse.py`, and create a file named `index.html` in that folder, with the following content:

```
<!DOCTYPE html>
<html>
<head>
  <title>Flask-SSE Quickstart</title>
</head>
<body>
  <h1>Flask-SSE Quickstart</h1>
  <script>
    var source = new EventSource("{{ url_for('sse.stream') }}");
    source.addEventListener('greeting', function(event) {
        var data = JSON.parse(event.data);
        alert("The server says " + data.message);
    }, false);
    source.addEventListener('error', function(event) {
        alert("Failed to connect to event stream. Is Redis running?");
    }, false);
  </script>
</body>
</html>
```

Run your code using gunicorn's gevent workers:

```
$ gunicorn sse:app --worker-class gevent --bind 127.0.0.1:8000
```

Open your web browser, and visit `127.0.0.1:8000`. Your web browser will automatically connect to the server-sent event stream. Open another tab, and visit `127.0.0.1:8000/hello`. You should get a Javascript alert in the first tab when you do so.

CHAPTER 2

Configuration

2.1 Redis

In order to use Flask-SSE, you need a Redis server to handle pubsub. Flask-SSE will search the application config for a Redis connection URL to use. It will try the following configuration values, in order:

1. `SSE_REDIS_URL`
2. `REDIS_URL`

If it doesn't find a Redis connection URL, Flask-SSE will raise a `KeyError` any time a client tries to access the SSE stream, or any time an event is published.

We recommend that you set this connection URL in an environment variable, and then load it into your application configuration using `os.environ`, like this:

```
import os
from flask import Flask

app = Flask(__name__)
app.config["REDIS_URL"] = os.environ.get("REDIS_URL")
```

If you are using a Redis server that has a password use:

```
app.config["REDIS_URL"] = "redis://:password@localhost"
```

2.2 Application Server

Flask-SSE does *not* work with Flask's built-in development server, due to the nature of the server-sent events protocol. This protocol uses long-lived HTTP requests to push data from the server to the client, which means that an HTTP request to the event stream will effectively never complete. Flask's built-in development server is single threaded, so it can only handle one HTTP request at a time. Once a client connects to the event stream, it will not be able to make any other HTTP requests to your site.

Instead, you must use a web server with asychronous workers. Asynchronous workers allow one worker to continuously handle the long-lived HTTP request that server-sent events require, while other workers simultaneously handle other HTTP requests to the server. Gunicorn is an excellent choice for an application server, since it can work with gevent to use asychronous workers: see gunicorn's design documentation.

For further information, see Flask's deployment documentation. Note that Flask's development server should **never** be used for deployment, regardless of whether you use Flask-SSE.

CHAPTER 3

Advanced Usage

3.1 Channels

Sometimes, you may not want all events to be published to all clients. For example, a client that cares about receiving the latest updates in their social network probably doesn't care about receiving the latest statistics about how many users are online across the entire site, and vice versa. When publishing an event, you can select which channel to direct the event to. If you do, only clients that are checking that particular channel will receive the event. For example, this event will be sent to the "users.social" channel:

```
sse.publish({"user": "alice", "status": "Life is good!"}, channel="users.social")
```

And this event will be sent to the "analytics" channel:

```
sse.publish({"active_users": 100}, channel="analytics")
```

Channel names can be any string you want, and are created dynamically as soon as they are referenced. The default channel name that Flask-SSE uses is "sse". For more information, see the documentation for the Redis pubsub system.

To subscribe to a channel, the client only needs to provide a `channel` query parameter when connecting to the event stream. For example, if your event stream is at `/stream`, you can connect to the "users.social" channel by using the URL `/stream?channel=users.social`. You can also use Flask's `url_for()` function to generate this query parameter, like so:

```
url_for("sse.stream", channel="users.social")
```

By default, all channels are publicly accessible to all users. However, see the next section to change that.

3.2 Access Control

Since Flask-SSE is implemented as a blueprint, you can attach a `before_request()` handler to implement access control. For example:

```
@sse.before_request
def check_access():
    if request.args.get("channel") == "analytics" and not g.user.is_admin():
        abort(403)

app.register_blueprint(sse, url_prefix='/sse')
```

Warning: When defining a `before_request()` handler, the blueprint must be registered *after* the handler is defined! Otherwise, the handler will have no effect.

CHAPTER 4

API

class flask_sse.**Message**(*data*, *type=None*, *id=None*, *retry=None*)

Data that is published as a server-sent event.

__init__(*data*, *type=None*, *id=None*, *retry=None*)

Create a server-sent event.

Parameters

- **data** – The event data. If it is not a string, it will be serialized to JSON using the Flask application's JSONEncoder.
- **type** – An optional event type.
- **id** – An optional event ID.
- **retry** – An optional integer, to specify the reconnect time for disconnected clients of this stream.

__str__()

Serialize this object to a string, according to the server-sent events specification.

to_dict()

Serialize this object to a minimal dictionary, for storing in Redis.

class flask_sse.**ServerSentEventsBlueprint**(*name: str*, *import_name: str*, *static_folder: Optional[str] = None*, *static_url_path: Optional[str] = None*, *template_folder: Optional[str] = None*, *url_prefix: Optional[str] = None*, *subdomain: Optional[str] = None*, *url_defaults: Optional[dict] = None*, *root_path: Optional[str] = None*, *cli_group: Optional[str] = <object object>*)

A flask.Blueprint subclass that knows how to publish, subscribe to, and stream server-sent events.

messages(*channel='sse'*)

A generator of *Message* objects from the given channel.

publish(*data*, *type=None*, *id=None*, *retry=None*, *channel='sse'*)
Publish data as a server-sent event.

Parameters

- **data** – The event data. If it is not a string, it will be serialized to JSON using the Flask application's `JSONEncoder`.
- **type** – An optional event type.
- **id** – An optional event ID.
- **retry** – An optional integer, to specify the reconnect time for disconnected clients of this stream.
- **channel** – If you want to direct different events to different clients, you may specify a channel for this event to go to. Only clients listening to the same channel will receive this event. Defaults to "sse".

redis
A `redis.StrictRedis` instance, configured to connect to the current application's Redis server.

stream()
A view function that streams server-sent events. Ignores any *`Last-Event-ID`* headers in the HTTP request. Use a "channel" query parameter to stream events from a different channel than the default channel (which is "sse").

flask_sse.**sse = <ServerSentEventsBlueprint 'sse'>**
An instance of *`ServerSentEventsBlueprint`* that hooks up the *`stream()`* method as a view function at the root of the blueprint. If you don't want to customize this blueprint at all, you can simply import and use this instance in your application.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

Symbols

__init__() (*flask_sse.Message method*), 9
__str__() (*flask_sse.Message method*), 9

M

Message (*class in flask_sse*), 9
messages() (*flask_sse.ServerSentEventsBlueprint method*), 9

P

publish() (*flask_sse.ServerSentEventsBlueprint method*), 9

R

redis (*flask_sse.ServerSentEventsBlueprint attribute*), 10

S

ServerSentEventsBlueprint (*class in flask_sse*), 9
sse (*in module flask_sse*), 10
stream() (*flask_sse.ServerSentEventsBlueprint method*), 10

T

to_dict() (*flask_sse.Message method*), 9