Deployment

There are many options to deploy a Flask-SocketIO server, ranging from simple to the insanely complex. In this section, the most commonly used options are described.

Embedded Server

The simplest deployment strategy is to start the web server by calling `socketio.run(app)` as shown in examples above. This will look through the packages that are installed for the best available web server start the application on it. The current web server choices that are evaluated are `eventlet`, `gevent` and the Flask development server.

If eventlet or gevent are available, `socketio.run(app)` starts a production-ready server using one of these frameworks. If neither of these are installed, then the Flask development web server is used, and in this case the server is not intended to be used in a production deployment.

Unfortunately this option is not available when using gevent with uWSGI. See the uWSGI section below for information on this option.

Gunicorn Web Server

An alternative to `socketio.run(app)` is to use gunicorn as web server, using the eventlet or gevent workers. For this option, eventlet or gevent need to be installed, in addition to gunicorn. The command line that starts the eventlet server via gunicorn is:

```
gunicorn --worker-class eventlet -w 1 module:app
```

If you prefer to use gevent, the command to start the server is:

```
gunicorn -k gevent -w 1 module:app
```

When using gunicorn with the gevent worker and the WebSocket support provided by gevent-websocket, the command that starts the server must be changed to select a custom gevent web server that supports the WebSocket protocol. The modified command is:

```
gunicorn -k geventwebsocket.gunicorn.workers.GeventWebSocketWorker -w 1 module:app
```

A third option with Gunicorn is to use the threaded worker, along with the simple-websocket package for WebSocket support. This is a particularly good solution for applications that are CPU heavy or are otherwise incompatible with eventlet and gevent use of green threads. The command to start a threaded web server is:

```
gunicorn -w 1 --threads 100 module:app
```

In all these commands, `module` is the Python module or package that defines the application instance, and `app` is the application instance itself.

Due to the limited load balancing algorithm used by gunicorn, it is not possible to use more than one worker process when using this web server. For that reason, all the examples above include the `-w 1` option.

The workaround to use multiple worker processes with gunicorn is to launch several single-worker instances and put them behind a more capable load balancer such as nginx.

uWSGI Web Server

When using the uWSGI server in combination with gevent, the Socket.IO server can take advantage of uWSGI's native WebSocket support.

A complete explanation of the configuration and usage of the uWSGI server is beyond the scope of this documentation. The uWSGI server is a fairly complex package that provides a large and comprehensive set

of options. It must be compiled with WebSocket and SSL support for the WebSocket transport to be available. As way of an introduction, the following command starts a uWSGI server for the example application app.py on port 5000:

```
$ uwsgi --http :5000 --gevent 1000 --http-websockets --master --wsgi-file app.py --callable ap
```

Using nginx as a WebSocket Reverse Proxy

It is possible to use nginx as a front-end reverse proxy that passes requests to the application. However, only releases of nginx 1.4 and newer support proxying of the WebSocket protocol. Below is a basic nginx configuration that proxies HTTP and WebSocket requests:

```
server {
    listen 80;
    server_name _;

    location / {
        include proxy_params;
        proxy_pass http://127.0.0.1:5000;
    }

    location /static/ {
        alias <path-to-your-application>/static/;
        expires 30d;
    }

    location /socket.io {
        include proxy_params;
        proxy_http_version 1.1;
        proxy_buffering off;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
        proxy_pass http://127.0.0.1:5000/socket.io;
    }
}
```

The next example adds the support for load balancing multiple Socket.IO servers:

```
upstream socketio_nodes {
    ip_hash;

    server 127.0.0.1:5000;
    server 127.0.0.1:5001;
    server 127.0.0.1:5002;
    # to scale the app, just add more nodes here!
}

server {
    listen 80;
    server_name _;

    location / {
        include proxy_params;
        proxy_pass http://127.0.0.1:5000;
    }

    location /static/ {
        alias <path-to-your-application>/static/;
        expires 30d;
    }

    location /socket.io {
        include proxy_params;
        proxy_http_version 1.1;
        proxy_buffering off;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
        proxy_pass http://socketio_nodes/socket.io;
    }
}
```

While the above examples can work as an initial configuration, be aware that a production install of nginx will need a more complete configuration covering other deployment aspects such as SSL support.

Using Multiple Workers

Flask-SocketIO supports multiple workers behind a load balancer starting with release 2.0. Deploying multiple workers gives applications that use Flask-SocketIO the ability to spread the client connections among multiple processes and hosts, and in this way scale to support very large numbers of concurrent clients.

There are two requirements to use multiple Flask-SocketIO workers:

- The load balancer must be configured to forward all HTTP requests from a given client always to the same worker. This is sometimes referenced as "sticky sessions". For nginx, use the `ip_hash` directive to achieve this. Gunicorn cannot be used with multiple workers because its load balancer algorithm does not support sticky sessions.
- Since each of the servers owns only a subset of the client connections, a message queue such as Redis or RabbitMQ is used by the servers to coordinate complex operations such as broadcasting and rooms.

When working with a message queue, there are additional dependencies that need to be installed:

- For Redis, the package `redis` must be installed (`pip install redis`).
- For RabbitMQ, the package `kombu` must be installed (`pip install kombu`).
- For Kafka, the package `kafka-python` must be installed (`pip install kafka-python`).
- For other message queues supported by Kombu, see the Kombu documentation to find out what dependencies are needed.
- If eventlet or gevent are used, then monkey patching the Python standard library is normally required to force the message queue package to use coroutine friendly functions and classes.

For eventlet, monkey patching is done with:

```
import eventlet
eventlet.monkey_patch()
```

For gevent, you can monkey patch the standard library with:

```
from gevent import monkey
monkey.patch_all()
```

In both cases it is recommended that you apply the monkey patching at the top of your main script, even above your imports.

To start multiple Flask-SocketIO servers, you must first ensure you have the message queue service running. To start a Socket.IO server and have it connect to the message queue, add the `message_queue` argument to the `SocketIO` constructor:

```
socketio = SocketIO(app, message_queue='redis://')
```

The value of the `message_queue` argument is the connection URL of the queue service that is used. For a redis queue running on the same host as the server, the `'redis://'` URL can be used. Likewise, for a default RabbitMQ queue the `'amqp://'` URL can be used. For Kafka, use a `kafka://` URL. The Kombu package has a documentation section that describes the format of the URLs for all the supported queues.

Emitting from an External Process

For many types of applications, it is necessary to emit events from a process that is not the SocketIO server, for an example a Celery worker. If the SocketIO server or servers are configured to listen on a message queue as shown in the previous section, then any other process can create its own `SocketIO` instance and use it to emit events in the same way the server does.

For example, for an application that runs on an eventlet web server and uses a Redis message queue, the

following Python script broadcasts an event to all clients:

```
socketio = SocketIO(message_queue='redis://')
socketio.emit('my event', {'data': 'foo'}, namespace='/test')
```

When using the `SocketIO` instance in this way, the Flask application instance is not passed to the constructor.

The `channel` argument to `SocketIO` can be used to select a specific channel of communication through the message queue. Using a custom channel name is necessary when there are multiple independent SocketIO services sharing the same queue.

Flask-SocketIO does not apply monkey patching when eventlet or gevent are used. But when working with a message queue, it is very likely that the Python package that talks to the message queue service will hang if the Python standard library is not monkey patched.

It is important to note that an external process that wants to connect to a SocketIO server does not need to use eventlet or gevent like the main server. Having a server use a coroutine framework, while an external process is not a problem. For example, Celery workers do not need to be configured to use eventlet or gevent just because the main server does. But if your external process does use a coroutine framework for whatever reason, then monkey patching is likely required, so that the message queue accesses coroutine friendly functions and classes.

Cross-Origin Controls

For security reasons, this server enforces a same-origin policy by default. In practical terms, this means the following:

- If an incoming HTTP or WebSocket request includes the `Origin` header, this header must match the scheme and host of the connection URL. In case of a mismatch, a 400 status code response is returned and the connection is rejected.
- No restrictions are imposed on incoming requests that do not include the `Origin` header.

If necessary, the `cors_allowed_origins` option can be used to allow other origins. This argument can be set to a string to set a single allowed origin, or to a list to allow multiple origins. A special value of `'*'` can be used to instruct the server to allow all origins, but this should be done with care, as this could make the server vulnerable to Cross-Site Request Forgery (CSRF) attacks.