flask-socketio Documentation

Miguel Grinberg

Apr 24, 2022

CONTENTS

Flask-SocketIO gives Flask applications access to low latency bi-directional communications between the clients and the server. The client-side application can use any of the SocketIO client libraries in Javascript, Python, C++, Java and Swift, or any other compatible client to establish a permanent connection to the server.

CHAPTER

ONE

INTRODUCTION

1.1 Installation

You can install this package in the usual way using `pip`:

```
pip install flask-socketio
```

1.2 Requirements

Flask-SocketIO is compatible with Python 3.6+. The asynchronous services that this package relies on can be selected among three choices:

- eventlet is the best performant option, with support for long-polling and WebSocket transports.
- gevent is supported in a number of different configurations. The long-polling transport is fully supported with the gevent package, but unlike eventlet, gevent does not have native WebSocket support. To add support for WebSocket there are currently two options. Installing the gevent-websocket package adds WebSocket support to gevent or one can use the uWSGI web server, which comes with WebSocket functionality. The use of gevent is also a performant option, but slightly lower than eventlet.
- The Flask development server based on Werkzeug can be used as well, with the caveat that this web server is intended only for development use, so it should only be used to simplify the development workflow and not for production.

The extension automatically detects which asynchronous framework to use based on what is installed. Preference is given to eventlet, followed by gevent. For WebSocket support in gevent, uWSGI is preferred, followed by gevent-websocket. If neither eventlet nor gevent are installed, then the Flask development server is used.

If using multiple processes, a message queue service must be configured to allow the servers to coordinate operations such as broadcasting. The supported queues are Redis, RabbitMQ, Kafka, and any other message queues supported by the Kombu package.

On the client-side, the official Socket.IO Javascript client library can be used to establish a connection to the server. There are also official clients written in Swift, Java and C++. Unofficial clients may also work, as long as they implement the Socket.IO protocol. The python-socketio package (which provides the Socket.IO server implementation used by Flask-SocketIO) includes a Python client.

1.3 Version compatibility

The Socket.IO protocol has been through a number of revisions, and some of these introduced backward incompatible changes, which means that the client and the server must use compatible versions for everything to work.

The version compatibility chart below maps versions of this package to versions of the JavaScript reference implementation and the versions of the Socket.IO and Engine.IO protocols.

JavaScript Socket.IO version	Socket.IO protocol revision	Engine.IO protocol revision	Flask-SocketIO version	python-socketio version	python-engineio version
0.9.x	1, 2	1, 2	Not supported	Not supported	Not supported
1.x and 2.x	3, 4	3	4.x	4.x	3.x
3.x and 4.x	5	4	5.x	5.x	4.x

CHAPTER

TWO

GETTING STARTED

2.1 Initialization

The following code example shows how to add Flask-SocketIO to a Flask application:

```
from flask import Flask, render_template
from flask_socketio import SocketIO

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret!'
socketio = SocketIO(app)

if __name__ == '__main__':
    socketio.run(app)
```

The `init_app()` style of initialization is also supported. To start the web server simply execute your script. Note the way the web server is started. The `socketio.run()` function encapsulates the start up of the web server and replaces the `app.run()` standard Flask development server start up. When the application is in debug mode the Werkzeug development server is still used and configured properly inside `socketio.run()`. In production mode the eventlet web server is used if available, else the gevent web server is used. If eventlet and gevent are not installed, the Werkzeug development web server is used.

The `flask run` command introduced in Flask 0.11 can be used to start a Flask-SocketIO development server based on Werkzeug, but this method of starting the Flask-SocketIO server is not recommended due to lack of WebSocket support. Previous versions of this package included a customized version of the `flask run` command that allowed the use of WebSocket on eventlet and gevent production servers, but this functionality has been discontinued in favor of the `socketio.run(app)` startup method shown above which is more robust.

The application must serve a page to the client that loads the Socket.IO library and establishes a connection:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.0.1/socket.io.js"␣
↪integrity="sha512-q/
↪dWJ3kcmjBLU4Qc47E4A9kTB4m3wuTY7vkFJDTZKjTs8jhyGQnaUrxa0Ytd0ssMZhbNua9hE+E7Qv1j+DyZwA==
↪" crossorigin="anonymous"></script>
<script type="text/javascript" charset="utf-8">
    var socket = io();
    socket.on('connect', function() {
        socket.emit('my event', {data: 'I\'m connected!'});
    });
</script>
```

2.2 Receiving Messages

When using SocketIO, messages are received by both parties as events. On the client side Javascript callbacks are used. With Flask-SocketIO the server needs to register handlers for these events, similarly to how routes are handled by view functions.

The following example creates a server-side event handler for an unnamed event:

```
@socketio.on('message')
def handle_message(data):
    print('received message: ' + data)
```

The above example uses string messages. Another type of unnamed events use JSON data:

```
@socketio.on('json')
def handle_json(json):
    print('received json: ' + str(json))
```

The most flexible type of event uses custom event names. The message data for these events can be string, bytes, int, or JSON:

```
@socketio.on('my event')
def handle_my_custom_event(json):
    print('received json: ' + str(json))
```

Custom named events can also support multiple arguments:

```
@socketio.on('my_event')
def handle_my_custom_event(arg1, arg2, arg3):
    print('received args: ' + arg1 + arg2 + arg3)
```

When the name of the event is a valid Python identifier that does not collide with other defined symbols, the `@socketio.event` decorator provides a more compact syntax that takes the event name from the decorated function:

```
@socketio.event
def my_custom_event(arg1, arg2, arg3):
    print('received args: ' + arg1 + arg2 + arg3)
```

Named events are the most flexible, as they eliminate the need to include additional metadata to describe the message type. The names `message`, `json`, `connect` and `disconnect` are reserved and cannot be used for named events.

Flask-SocketIO also supports SocketIO namespaces, which allow the client to multiplex several independent connections on the same physical socket:

```
@socketio.on('my event', namespace='/test')
def handle_my_custom_namespace_event(json):
    print('received json: ' + str(json))
```

When a namespace is not specified a default global namespace with the name `'/'` is used.

For cases when a decorator syntax isn't convenient, the `on_event` method can be used:

```
def my_function_handler(data):
    pass

socketio.on_event('my event', my_function_handler, namespace='/test')
```

Clients may request an acknowledgement callback that confirms receipt of a message they sent. Any values returned from the handler function will be passed to the client as arguments in the callback function:

```
@socketio.on('my event')
def handle_my_custom_event(json):
    print('received json: ' + str(json))
    return 'one', 2
```

In the above example, the client callback function will be invoked with two arguments, `'one'` and `2`. If a handler function does not return any values, the client callback function will be invoked without arguments.

2.3 Sending Messages

SocketIO event handlers defined as shown in the previous section can send reply messages to the connected client using the `send()` and `emit()` functions.

The following examples bounce received events back to the client that sent them:

```
from flask_socketio import send, emit

@socketio.on('message')
def handle_message(message):
    send(message)

@socketio.on('json')
def handle_json(json):
    send(json, json=True)

@socketio.on('my event')
def handle_my_custom_event(json):
    emit('my response', json)
```

Note how `send()` and `emit()` are used for unnamed and named events respectively.

When working with namespaces, `send()` and `emit()` use the namespace of the incoming message by default. A different namespace can be specified with the optional `namespace` argument:

```
@socketio.on('message')
def handle_message(message):
    send(message, namespace='/chat')

@socketio.on('my event')
def handle_my_custom_event(json):
    emit('my response', json, namespace='/chat')
```

To send an event with multiple arguments, send a tuple:

```
@socketio.on('my event')
def handle_my_custom_event(json):
    emit('my response', ('foo', 'bar', json), namespace='/chat')
```

SocketIO supports acknowledgment callbacks that confirm that a message was received by the client:

```
def ack():
    print('message was received!')

@socketio.on('my event')
def handle_my_custom_event(json):
    emit('my response', json, callback=ack)
```

When using callbacks, the Javascript client receives a callback function to invoke upon receipt of the message. After the client application invokes the callback function the server invokes the corresponding server-side callback. If the client-side callback is invoked with arguments, these are provided as arguments to the server-side callback as well.

2.4 Broadcasting

Another very useful feature of SocketIO is the broadcasting of messages. Flask-SocketIO supports this feature with the `broadcast=True` optional argument to `send()` and `emit()`:

```
@socketio.on('my event')
def handle_my_custom_event(data):
    emit('my response', data, broadcast=True)
```

When a message is sent with the broadcast option enabled, all clients connected to the namespace receive it, including the sender. When namespaces are not used, the clients connected to the global namespace receive the message. Note that callbacks are not invoked for broadcast messages.

In all the examples shown until this point the server responds to an event sent by the client. But for some applications, the server needs to be the originator of a message. This can be useful to send notifications to clients of events that originated in the server, for example in a background thread. The `socketio.send()` and `socketio.emit()` methods can be used to broadcast to all connected clients:

```
def some_function():
    socketio.emit('some event', {'data': 42})
```

Note that `socketio.send()` and `socketio.emit()` are not the same functions as the context-aware `send()` and `emit()`. Also note that in the above usage there is no client context, so `broadcast=True` is assumed and does not need to be specified.

2.5 Rooms

For many applications it is necessary to group users into subsets that can be addressed together. The best example is a chat application with multiple rooms, where users receive messages from the room or rooms they are in, but not from other rooms where other users are. Flask-SocketIO supports this concept of rooms through the `join_room()` and `leave_room()` functions:

```
from flask_socketio import join_room, leave_room

@socketio.on('join')
def on_join(data):
    username = data['username']
    room = data['room']
    join_room(room)
```

(continues on next page)

(continued from previous page)

```
    send(username + ' has entered the room.', to=room)

@socketio.on('leave')
def on_leave(data):
    username = data['username']
    room = data['room']
    leave_room(room)
    send(username + ' has left the room.', to=room)
```

The `send()` and `emit()` functions accept an optional `to` argument that cause the message to be sent to all the clients that are in the given room.

All clients are assigned a room when they connect, named with the session ID of the connection, which can be obtained from `request.sid`. A given client can join any rooms, which can be given any names. When a client disconnects it is removed from all the rooms it was in. The context-free `socketio.send()` and `socketio.emit()` functions also accept a `to` argument to broadcast to all clients in a room.

Since all clients are assigned a personal room, to address a message to a single client, the session ID of the client can be used as the `to` argument.

2.6 Connection Events

Flask-SocketIO also dispatches connection and disconnection events. The following example shows how to register handlers for them:

```
@socketio.on('connect')
def test_connect(auth):
    emit('my response', {'data': 'Connected'})

@socketio.on('disconnect')
def test_disconnect():
    print('Client disconnected')
```

The `auth` argument in the connection handler is optional. The client can use it to pass authentication data such as tokens in dictionary format. If the client does not provide authentication details, then this argument is set to `None`. If the server defines a connection event handler without this argument, then any authentication data passed by the cient is discarded.

The connection event handler can return `False` to reject the connection, or it can also raise *ConnectionRefusedError*. This is so that the client can be authenticated at this point. When using the exception, any arguments passed to the exception are returned to the client in the error packet. Examples:

```
from flask_socketio import ConnectionRefusedError

@socketio.on('connect')
def connect():
    if not self.authenticate(request.args):
        raise ConnectionRefusedError('unauthorized!')
```

Note that connection and disconnection events are sent individually on each namespace used.

2.7 Class-Based Namespaces

As an alternative to the decorator-based event handlers described above, the event handlers that belong to a namespace can be created as methods of a class. The *flask_socketio.Namespace* is provided as a base class to create class-based namespaces:

```
from flask_socketio import Namespace, emit

class MyCustomNamespace(Namespace):
    def on_connect(self):
        pass

    def on_disconnect(self):
        pass

    def on_my_event(self, data):
        emit('my_response', data)

socketio.on_namespace(MyCustomNamespace('/test'))
```

When class-based namespaces are used, any events received by the server are dispatched to a method named as the event name with the `on_` prefix. For example, event `my_event` will be handled by a method named `on_my_event`. If an event is received for which there is no corresponding method defined in the namespace class, then the event is ignored. All event names used in class-based namespaces must use characters that are legal in method names.

As a convenience to methods defined in a class-based namespace, the namespace instance includes versions of several of the methods in the *flask_socketio.SocketIO* class that default to the proper namespace when the `namespace` argument is not given.

If an event has a handler in a class-based namespace, and also a decorator-based function handler, only the decorated function handler is invoked.

2.8 Error Handling

Flask-SocketIO can also deal with exceptions:

```
@socketio.on_error()        # Handles the default namespace
def error_handler(e):
    pass

@socketio.on_error('/chat') # handles the '/chat' namespace
def error_handler_chat(e):
    pass

@socketio.on_error_default  # handles all namespaces without an explicit error handler
def default_error_handler(e):
    pass
```

Error handler functions take the exception object as an argument.

The message and data arguments of the current request can also be inspected with the `request.event` variable, which is useful for error logging and debugging outside the event handler:

```
from flask import request

@socketio.on("my error event")
def on_my_event(data):
    raise RuntimeError()

@socketio.on_error_default
def default_error_handler(e):
    print(request.event["message"]) # "my error event"
    print(request.event["args"])    # (data,)
```

2.9 Debugging and Troubleshooting

To help you debug issues, the server can be configured to output logs to the terminal:

```
socketio = SocketIO(logger=True, engineio_logger=True)
```

The `logger` argument controls logging related to the Socket.IO protocol, while `engineio_logger` controls logs that originate in the low-level Engine.IO transport. These arguments can be set to `True` to output logs to `stderr`, or to an object compatible with Python's `logging` package where the logs should be emitted to. A value of `False` disables logging.

Logging can help identify the cause of connection problems, 400 responses, bad performance and other issues.

CHAPTER

THREE

IMPLEMENTATION NOTES

3.1 Access to Flask's Context Globals

Handlers for SocketIO events are different than handlers for routes and that introduces a lot of confusion around what can and cannot be done in a SocketIO handler. The main difference is that all the SocketIO events generated for a client occur in the context of a single long running request.

In spite of the differences, Flask-SocketIO attempts to make working with SocketIO event handlers easier by making the environment similar to that of a regular HTTP request. The following list describes what works and what doesn't:

- An application context is pushed before invoking an event handler making `current_app` and `g` available to the handler.
- A request context is also pushed before invoking a handler, also making `request` and `session` available. But note that WebSocket events do not have individual requests associated with them, so the request context that started the connection is pushed for all the events that are dispatched during the life of the connection.
- The `request` context global is enhanced with a `sid` member that is set to a unique session ID for the connection. This value is used as an initial room where the client is added.
- The `request` context global is enhanced with `namespace` and `event` members that contain the currently handled namespace and event arguments. The `event` member is a dictionary with `message` and `args` keys.
- The `session` context global behaves in a different way than in regular requests. A copy of the user session at the time the SocketIO connection is established is made available to handlers invoked in the context of that connection. If a SocketIO handler modifies the session, the modified session will be preserved for future SocketIO handlers, but regular HTTP route handlers will not see these changes. Effectively, when a SocketIO handler modifies the session, a "fork" of the session is created exclusively for these handlers. The technical reason for this limitation is that to save the user session a cookie needs to be sent to the client, and that requires HTTP request and response, which do not exist in a SocketIO connection. When using server-side sessions such as those provided by the Flask-Session or Flask-KVSession extensions, changes made to the session in HTTP route handlers can be seen by SocketIO handlers, as long as the session is not modified in the SocketIO handlers.
- The `before_request` and `after_request` hooks are not invoked for SocketIO event handlers.
- SocketIO handlers can take custom decorators, but most Flask decorators will not be appropriate to use for a SocketIO handler, given that there is no concept of a `Response` object during a SocketIO connection.

3.2 Authentication

A common need of applications is to validate the identity of their users. The traditional mechanisms based on web forms and HTTP requests cannot be used in a SocketIO connection, since there is no place to send HTTP requests and responses. If necessary, an application can implement a customized login form that sends credentials to the server as a SocketIO message when the submit button is pressed by the user.

However, in most cases it is more convenient to perform the traditional authentication process before the SocketIO connection is established. The user's identity can then be recorded in the user session or in a cookie, and later when the SocketIO connection is established that information will be accessible to SocketIO event handlers.

Recent revisions of the Socket.IO protocol include the ability to pass a dictionary with authentication information during the connection. This is an ideal place for the client to include a token or other authentication details. If the client uses this capability, the server will provide this dictionary as an argument to the `connect` event handler, as shown above.

3.2.1 Using Flask-Login with Flask-SocketIO

Flask-SocketIO can access login information maintained by Flask-Login. After a regular Flask-Login authentication is performed and the `login_user()` function is called to record the user in the user session, any SocketIO connections will have access to the `current_user` context variable:

```
@socketio.on('connect')
def connect_handler():
    if current_user.is_authenticated:
        emit('my response',
             {'message': '{0} has joined'.format(current_user.name)},
             broadcast=True)
    else:
        return False  # not allowed here
```

Note that the `login_required` decorator cannot be used with SocketIO event handlers, but a custom decorator that disconnects non-authenticated users can be created as follows:

```
import functools
from flask import request
from flask_login import current_user
from flask_socketio import disconnect, emit

def authenticated_only(f):
    @functools.wraps(f)
    def wrapped(*args, **kwargs):
        if not current_user.is_authenticated:
            disconnect()
        else:
            return f(*args, **kwargs)
    return wrapped

@socketio.on('my event')
@authenticated_only
def handle_my_custom_event(data):
    emit('my response', {'message': '{0} has joined'.format(current_user.name)},
         broadcast=True)
```

CHAPTER

FOUR

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.

4.1 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.

4.2 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.

4.3 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 app
```

4.4 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;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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.

4.5 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.

4.6 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.

4.7 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.

CHAPTER

FIVE

UPGRADING TO FLASK-SOCKETIO 5.X FROM THE 4.X RELEASES

The Socket.IO protocol recently introduced a series of backwards incompatible changes. The 5.x releases of Flask-SocketIO adopted these changes, and for that reason it can only be used with clients that have also been updated to the current version of the protocol. In particular, this means that the JavaScript client must be upgraded to a 3.x release, and if your client hasn't been upgraded to the latest version of the Socket.IO protocol, then you must use a Flask-SocketIO 4.x release.

The following protocol changes are of importance, as they may affect existing applications:

- The default namespace `'/'` is not automatically connected anymore, and is now treated in the same way as other namespaces.
- Each namespace connection has its own `sid` value, different from the others and different from the Engine.IO `sid`.
- Flask-SocketIO now uses the same ping interval and timeout values as the JavaScript reference implementation, which are 25 and 5 seconds respectively.
- The ping/pong mechanism has been reversed. In the current version of the protocol, the server issues a ping and the client responds with a pong.
- The default allowed payload size for long–polling packets has been lowered from 100MB to 1MB.
- The *io* cookie is not sent to the client anymore by default.

CHAPTER

SIX

API REFERENCE

class flask_socketio.**SocketIO**(*app=None*, ***kwargs*)

Create a Flask-SocketIO server.

Parameters

- **app** – The flask application instance. If the application instance isn't known at the time this class is instantiated, then call `socketio.init_app(app)` once the application instance is available.
- **manage_session** – If set to `True`, this extension manages the user session for Socket.IO events. If set to `False`, Flask's own session management is used. When using Flask's cookie based sessions it is recommended that you leave this set to the default of `True`. When using server-side sessions, a `False` setting enables sharing the user session between HTTP routes and Socket.IO events.
- **message_queue** – A connection URL for a message queue service the server can use for multi-process communication. A message queue is not required when using a single server process.
- **channel** – The channel name, when using a message queue. If a channel isn't specified, a default channel will be used. If multiple clusters of SocketIO processes need to use the same message queue without interfering with each other, then each cluster should use a different channel.
- **path** – The path where the Socket.IO server is exposed. Defaults to `'socket.io'`. Leave this as is unless you know what you are doing.
- **resource** – Alias to `path`.
- **kwargs** – Socket.IO and Engine.IO server options.

The Socket.IO server options are detailed below:

Parameters

- **client_manager** – The client manager instance that will manage the client list. When this is omitted, the client list is stored in an in-memory structure, so the use of multiple connected servers is not possible. In most cases, this argument does not need to be set explicitly.
- **logger** – To enable logging set to `True` or pass a logger object to use. To disable logging set to `False`. The default is `False`. Note that fatal errors will be logged even when `logger` is `False`.
- **json** – An alternative json module to use for encoding and decoding packets. Custom json modules must have `dumps` and `loads` functions that are compatible with the standard library versions. To use the same json encoder and decoder as a Flask application, use `flask.json`.

- **async_handlers** – If set to `True`, event handlers for a client are executed in separate threads. To run handlers for a client synchronously, set to `False`. The default is `True`.
- **always_connect** – When set to `False`, new connections are provisory until the connect handler returns something other than `False`, at which point they are accepted. When set to `True`, connections are immediately accepted, and then if the connect handler returns `False` a disconnect is issued. Set to `True` if you need to emit events from the connect handler and your client is confused when it receives events before the connection acceptance. In any other case use the default of `False`.

The Engine.IO server configuration supports the following settings:

Parameters

- **async_mode** – The asynchronous model to use. See the Deployment section in the documentation for a description of the available options. Valid async modes are `threading`, `eventlet`, `gevent` and `gevent_uwsgi`. If this argument is not given, `eventlet` is tried first, then `gevent_uwsgi`, then `gevent`, and finally `threading`. The first async mode that has all its dependencies installed is then one that is chosen.
- **ping_interval** – The interval in seconds at which the server pings the client. The default is 25 seconds. For advanced control, a two element tuple can be given, where the first number is the ping interval and the second is a grace period added by the server.
- **ping_timeout** – The time in seconds that the client waits for the server to respond before disconnecting. The default is 5 seconds.
- **max_http_buffer_size** – The maximum size of a message when using the polling transport. The default is 1,000,000 bytes.
- **allow_upgrades** – Whether to allow transport upgrades or not. The default is `True`.
- **http_compression** – Whether to compress packages when using the polling transport. The default is `True`.
- **compression_threshold** – Only compress messages when their byte size is greater than this value. The default is 1024 bytes.
- **cookie** – If set to a string, it is the name of the HTTP cookie the server sends back to the client containing the client session id. If set to a dictionary, the `'name'` key contains the cookie name and other keys define cookie attributes, where the value of each attribute can be a string, a callable with no arguments, or a boolean. If set to `None` (the default), a cookie is not sent to the client.
- **cors_allowed_origins** – Origin or list of origins that are allowed to connect to this server. Only the same origin is allowed by default. Set this argument to `'*'` to allow all origins, or to `[]` to disable CORS handling.
- **cors_credentials** – Whether credentials (cookies, authentication) are allowed in requests to this server. The default is `True`.
- **monitor_clients** – If set to `True`, a background task will ensure inactive clients are closed. Set to `False` to disable the monitoring task (not recommended). The default is `True`.
- **engineio_logger** – To enable Engine.IO logging set to `True` or pass a logger object to use. To disable logging set to `False`. The default is `False`. Note that fatal errors are logged even when `engineio_logger` is `False`.

close_room(*room*, *namespace=None*)

Close a room.

This function removes any users that are in the given room and then deletes the room from the server. This function can be used outside of a SocketIO event context.

Parameters

- **room** – The name of the room to close.
- **namespace** – The namespace under which the room exists. Defaults to the global namespace.

emit(*event*, **args*, ***kwargs*)

Emit a server generated SocketIO event.

This function emits a SocketIO event to one or more connected clients. A JSON blob can be attached to the event as payload. This function can be used outside of a SocketIO event context, so it is appropriate to use when the server is the originator of an event, outside of any client context, such as in a regular HTTP request handler or a background task. Example:

```
@app.route('/ping')
def ping():
    socketio.emit('ping event', {'data': 42}, namespace='/chat')
```

Parameters

- **event** – The name of the user event to emit.
- **args** – A dictionary with the JSON data to send as payload.
- **namespace** – The namespace under which the message is to be sent. Defaults to the global namespace.
- **to** – Send the message to all the users in the given room. If this parameter is not included, the event is sent to all connected users.
- **include_self** – `True` to include the sender when broadcasting or addressing a room, or `False` to send to everyone but the sender.
- **skip_sid** – The session id of a client to ignore when broadcasting or addressing a room. This is typically set to the originator of the message, so that everyone except that client receive the message. To skip multiple sids pass a list.
- **callback** – If given, this function will be called to acknowledge that the client has received the message. The arguments that will be passed to the function are those provided by the client. Callback functions can only be used when addressing an individual client.

event(**args*, ***kwargs*)

Decorator to register an event handler.

This is a simplified version of the `on()` method that takes the event name from the decorated function.

Example usage:

```
@socketio.event
def my_event(data):
    print('Received data: ', data)
```

The above example is equivalent to:

```
@socketio.on('my_event')
def my_event(data):
    print('Received data: ', data)
```

A custom namespace can be given as an argument to the decorator:

```
@socketio.event(namespace='/test')
def my_event(data):
    print('Received data: ', data)
```

on(*message*, *namespace=None*)

Decorator to register a SocketIO event handler.

This decorator must be applied to SocketIO event handlers. Example:

```
@socketio.on('my event', namespace='/chat')
def handle_my_custom_event(json):
    print('received json: ' + str(json))
```

Parameters

- **message** – The name of the event. This is normally a user defined string, but a few event names are already defined. Use `'message'` to define a handler that takes a string payload, `'json'` to define a handler that takes a JSON blob payload, `'connect'` or `'disconnect'` to create handlers for connection and disconnection events.
- **namespace** – The namespace on which the handler is to be registered. Defaults to the global namespace.

on_error(*namespace=None*)

Decorator to define a custom error handler for SocketIO events.

This decorator can be applied to a function that acts as an error handler for a namespace. This handler will be invoked when a SocketIO event handler raises an exception. The handler function must accept one argument, which is the exception raised. Example:

```
@socketio.on_error(namespace='/chat')
def chat_error_handler(e):
    print('An error has occurred: ' + str(e))
```

Parameters **namespace** – The namespace for which to register the error handler. Defaults to the global namespace.

on_error_default(*exception_handler*)

Decorator to define a default error handler for SocketIO events.

This decorator can be applied to a function that acts as a default error handler for any namespaces that do not have a specific handler. Example:

```
@socketio.on_error_default
def error_handler(e):
    print('An error has occurred: ' + str(e))
```

on_event(*message*, *handler*, *namespace=None*)

Register a SocketIO event handler.

`on_event` is the non-decorator version of `'on'`.

Example:

```
def on_foo_event(json):
    print('received json: ' + str(json))

socketio.on_event('my event', on_foo_event, namespace='/chat')
```

Parameters

- **message** – The name of the event. This is normally a user defined string, but a few event names are already defined. Use `'message'` to define a handler that takes a string payload, `'json'` to define a handler that takes a JSON blob payload, `'connect'` or `'disconnect'` to create handlers for connection and disconnection events.
- **handler** – The function that handles the event.
- **namespace** – The namespace on which the handler is to be registered. Defaults to the global namespace.

run(*app*, *host=None*, *port=None*, ***kwargs*)

Run the SocketIO web server.

Parameters

- **app** – The Flask application instance.
- **host** – The hostname or IP address for the server to listen on. Defaults to 127.0.0.1.
- **port** – The port number for the server to listen on. Defaults to 5000.
- **debug** – `True` to start the server in debug mode, `False` to start in normal mode.
- **use_reloader** – `True` to enable the Flask reloader, `False` to disable it.
- **reloader_options** – A dictionary with options that are passed to the Flask reloader, such as `extra_files`, `reloader_type`, etc.
- **extra_files** – A list of additional files that the Flask reloader should watch. Defaults to `None`. Deprecated, use `reloader_options` instead.
- **log_output** – If `True`, the server logs all incoming connections. If `False` logging is disabled. Defaults to `True` in debug mode, `False` in normal mode. Unused when the threading async mode is used.
- **kwargs** – Additional web server options. The web server options are specific to the server used in each of the supported async modes. Note that options provided here will not be seen when using an external web server such as gunicorn, since this method is not called in that case.

send(*data*, *json=False*, *namespace=None*, *to=None*, *callback=None*, *include_self=True*, *skip_sid=None*, ***kwargs*)

Send a server-generated SocketIO message.

This function sends a simple SocketIO message to one or more connected clients. The message can be a string or a JSON blob. This is a simpler version of `emit()`, which should be preferred. This function can be used outside of a SocketIO event context, so it is appropriate to use when the server is the originator of an event.

Parameters

- **data** – The message to send, either a string or a JSON blob.
- **json** – `True` if `message` is a JSON blob, `False` otherwise.

- **namespace** – The namespace under which the message is to be sent. Defaults to the global namespace.
- **to** – Send the message only to the users in the given room. If this parameter is not included, the message is sent to all connected users.
- **include_self** – `True` to include the sender when broadcasting or addressing a room, or `False` to send to everyone but the sender.
- **skip_sid** – The session id of a client to ignore when broadcasting or addressing a room. This is typically set to the originator of the message, so that everyone except that client receive the message. To skip multiple sids pass a list.
- **callback** – If given, this function will be called to acknowledge that the client has received the message. The arguments that will be passed to the function are those provided by the client. Callback functions can only be used when addressing an individual client.

sleep(*seconds=0*)

Sleep for the requested amount of time using the appropriate async model.

This is a utility function that applications can use to put a task to sleep without having to worry about using the correct call for the selected async mode.

start_background_task(*target*, **args*, ***kwargs*)

Start a background task using the appropriate async model.

This is a utility function that applications can use to start a background task using the method that is compatible with the selected async mode.

Parameters

- **target** – the target function to execute.
- **args** – arguments to pass to the function.
- **kwargs** – keyword arguments to pass to the function.

This function returns an object that represents the background task, on which the `join()` methond can be invoked to wait for the task to complete.

stop()

Stop a running SocketIO web server.

This method must be called from a HTTP or SocketIO handler function.

test_client(*app*, *namespace=None*, *query_string=None*, *headers=None*, *auth=None*, *flask_test_client=None*)

The Socket.IO test client is useful for testing a Flask-SocketIO server. It works in a similar way to the Flask Test Client, but adapted to the Socket.IO server.

Parameters

- **app** – The Flask application instance.
- **namespace** – The namespace for the client. If not provided, the client connects to the server on the global namespace.
- **query_string** – A string with custom query string arguments.
- **headers** – A dictionary with custom HTTP headers.
- **auth** – Optional authentication data, given as a dictionary.

- **flask_test_client** – The instance of the Flask test client currently in use. Passing the Flask test client is optional, but is necessary if you want the Flask user session and any other cookies set in HTTP routes accessible from Socket.IO events.

flask_socketio.**emit**(*event*, **args*, ***kwargs*)

Emit a SocketIO event.

This function emits a SocketIO event to one or more connected clients. A JSON blob can be attached to the event as payload. This is a function that can only be called from a SocketIO event handler, as in obtains some information from the current client context. Example:

```
@socketio.on('my event')
def handle_my_custom_event(json):
    emit('my response', {'data': 42})
```

Parameters

- **event** – The name of the user event to emit.
- **args** – A dictionary with the JSON data to send as payload.
- **namespace** – The namespace under which the message is to be sent. Defaults to the namespace used by the originating event. A `'/'` can be used to explicitly specify the global namespace.
- **callback** – Callback function to invoke with the client's acknowledgement.
- **broadcast** – `True` to send the message to all clients, or `False` to only reply to the sender of the originating event.
- **to** – Send the message to all the users in the given room. If this argument is not set and `broadcast` is `False`, then the message is sent only to the originating user.
- **include_self** – `True` to include the sender when broadcasting or addressing a room, or `False` to send to everyone but the sender.
- **skip_sid** – The session id of a client to ignore when broadcasting or addressing a room. This is typically set to the originator of the message, so that everyone except that client receive the message. To skip multiple sids pass a list.
- **ignore_queue** – Only used when a message queue is configured. If set to `True`, the event is emitted to the clients directly, without going through the queue. This is more efficient, but only works when a single server process is used, or when there is a single addressee. It is recommended to always leave this parameter with its default value of `False`.

flask_socketio.**send**(*message*, ***kwargs*)

Send a SocketIO message.

This function sends a simple SocketIO message to one or more connected clients. The message can be a string or a JSON blob. This is a simpler version of `emit()`, which should be preferred. This is a function that can only be called from a SocketIO event handler.

Parameters

- **message** – The message to send, either a string or a JSON blob.
- **json** – `True` if `message` is a JSON blob, `False` otherwise.
- **namespace** – The namespace under which the message is to be sent. Defaults to the namespace used by the originating event. An empty string can be used to use the global namespace.
- **callback** – Callback function to invoke with the client's acknowledgement.

- **broadcast** – `True` to send the message to all connected clients, or `False` to only reply to the sender of the originating event.
- **to** – Send the message to all the users in the given room. If this argument is not set and `broadcast` is `False`, then the message is sent only to the originating user.
- **include_self** – `True` to include the sender when broadcasting or addressing a room, or `False` to send to everyone but the sender.
- **skip_sid** – The session id of a client to ignore when broadcasting or addressing a room. This is typically set to the originator of the message, so that everyone except that client receive the message. To skip multiple sids pass a list.
- **ignore_queue** – Only used when a message queue is configured. If set to `True`, the event is emitted to the clients directly, without going through the queue. This is more efficient, but only works when a single server process is used, or when there is a single addressee. It is recommended to always leave this parameter with its default value of `False`.

`flask_socketio.`**join_room**(*room*, *sid=None*, *namespace=None*)

Join a room.

This function puts the user in a room, under the current namespace. The user and the namespace are obtained from the event context. This is a function that can only be called from a SocketIO event handler. Example:

```
@socketio.on('join')
def on_join(data):
    username = session['username']
    room = data['room']
    join_room(room)
    send(username + ' has entered the room.', to=room)
```

Parameters

- **room** – The name of the room to join.
- **sid** – The session id of the client. If not provided, the client is obtained from the request context.
- **namespace** – The namespace for the room. If not provided, the namespace is obtained from the request context.

`flask_socketio.`**leave_room**(*room*, *sid=None*, *namespace=None*)

Leave a room.

This function removes the user from a room, under the current namespace. The user and the namespace are obtained from the event context. Example:

```
@socketio.on('leave')
def on_leave(data):
    username = session['username']
    room = data['room']
    leave_room(room)
    send(username + ' has left the room.', to=room)
```

Parameters

- **room** – The name of the room to leave.

- **sid** – The session id of the client. If not provided, the client is obtained from the request context.
- **namespace** – The namespace for the room. If not provided, the namespace is obtained from the request context.

flask_socketio.**close_room**(*room*, *namespace=None*)
Close a room.

This function removes any users that are in the given room and then deletes the room from the server.

Parameters

- **room** – The name of the room to close.
- **namespace** – The namespace for the room. If not provided, the namespace is obtained from the request context.

flask_socketio.**rooms**(*sid=None*, *namespace=None*)
Return a list of the rooms the client is in.

This function returns all the rooms the client has entered, including its own room, assigned by the Socket.IO server.

Parameters

- **sid** – The session id of the client. If not provided, the client is obtained from the request context.
- **namespace** – The namespace for the room. If not provided, the namespace is obtained from the request context.

flask_socketio.**disconnect**(*sid=None*, *namespace=None*, *silent=False*)
Disconnect the client.

This function terminates the connection with the client. As a result of this call the client will receive a disconnect event. Example:

```
@socketio.on('message')
def receive_message(msg):
    if is_banned(session['username']):
        disconnect()
    else:
        # ...
```

Parameters

- **sid** – The session id of the client. If not provided, the client is obtained from the request context.
- **namespace** – The namespace for the room. If not provided, the namespace is obtained from the request context.
- **silent** – this option is deprecated.

class flask_socketio.**Namespace**(*namespace=None*)

close_room(*room*, *namespace=None*)
Close a room.

emit(*event*, *data=None*, *room=None*, *include_self=True*, *namespace=None*, *callback=None*)
Emit a custom event to one or more connected clients.

send(*data*, *room=None*, *include_self=True*, *namespace=None*, *callback=None*)
Send a message to one or more connected clients.

trigger_event(*event*, **args*)
Dispatch an event to the proper handler method.

In the most common usage, this method is not overloaded by subclasses, as it performs the routing of events to methods. However, this method can be overridden if special dispatching rules are needed, or if having a single method that catches all events is desired.

class flask_socketio.**SocketIOTestClient**(*app*, *socketio*, *namespace=None*, *query_string=None*, *headers=None*, *auth=None*, *flask_test_client=None*)
This class is useful for testing a Flask-SocketIO server. It works in a similar way to the Flask Test Client, but adapted to the Socket.IO server.

Parameters

- **app** – The Flask application instance.
- **socketio** – The application's `SocketIO` instance.
- **namespace** – The namespace for the client. If not provided, the client connects to the server on the global namespace.
- **query_string** – A string with custom query string arguments.
- **headers** – A dictionary with custom HTTP headers.
- **auth** – Optional authentication data, given as a dictionary.
- **flask_test_client** – The instance of the Flask test client currently in use. Passing the Flask test client is optional, but is necessary if you want the Flask user session and any other cookies set in HTTP routes accessible from Socket.IO events.

connect(*namespace=None*, *query_string=None*, *headers=None*, *auth=None*)
Connect the client.

Parameters

- **namespace** – The namespace for the client. If not provided, the client connects to the server on the global namespace.
- **query_string** – A string with custom query string arguments.
- **headers** – A dictionary with custom HTTP headers.
- **auth** – Optional authentication data, given as a dictionary.

Note that it is usually not necessary to explicitly call this method, since a connection is automatically established when an instance of this class is created. An example where it this method would be useful is when the application accepts multiple namespace connections.

disconnect(*namespace=None*)
Disconnect the client.

Parameters **namespace** – The namespace to disconnect. The global namespace is assumed if this argument is not provided.

emit(*event*, **args*, ***kwargs*)
Emit an event to the server.

Parameters

- **event** – The event name.
- ***args** – The event arguments.

- **callback** – `True` if the client requests a callback, `False` if not. Note that client-side callbacks are not implemented, a callback request will just tell the server to provide the arguments to invoke the callback, but no callback is invoked. Instead, the arguments that the server provided for the callback are returned by this function.
- **namespace** – The namespace of the event. The global namespace is assumed if this argument is not provided.

get_received(*namespace=None*)
Return the list of messages received from the server.

Since this is not a real client, any time the server emits an event, the event is simply stored. The test code can invoke this method to obtain the list of events that were received since the last call.

Parameters **namespace** – The namespace to get events from. The global namespace is assumed if this argument is not provided.

is_connected(*namespace=None*)
Check if a namespace is connected.

Parameters **namespace** – The namespace to check. The global namespace is assumed if this argument is not provided.

send(*data*, *json=False*, *callback=False*, *namespace=None*)
Send a text or JSON message to the server.

Parameters

- **data** – A string, dictionary or list to send to the server.
- **json** – `True` to send a JSON message, `False` to send a text message.
- **callback** – `True` if the client requests a callback, `False` if not. Note that client-side callbacks are not implemented, a callback request will just tell the server to provide the arguments to invoke the callback, but no callback is invoked. Instead, the arguments that the server provided for the callback are returned by this function.
- **namespace** – The namespace of the event. The global namespace is assumed if this argument is not provided.

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX