socketio Documentation

Release 0.1

Miguel Grinberg

Oct 02, 2017

Contents

This project implements a Python Socket.IO server that can run standalone or integrated with a web application. The following are some of its features:

- Fully compatible with the Javascript, Swift, C++ and Java official Socket.IO clients, plus any third party clients that comply with the Socket.IO specification.
- Compatible with Python 2.7 and Python 3.3+.
- Supports large number of clients even on modest hardware when used with an asynchronous server based on asyncio (sanic and aiohttp), eventlet or gevent. For development and testing, any WSGI compliant multi-threaded server can also be used.
- Includes a WSGI middleware that integrates Socket.IO traffic with standard WSGI applications.
- Broadcasting of messages to all connected clients, or to subsets of them assigned to "rooms".
- Optional support for multiple servers, connected through a messaging queue such as Redis or RabbitMQ.
- Send messages to clients from external processes, such as Celery workers or auxiliary scripts.
- Event-based architecture implemented with decorators that hides the details of the protocol.
- Support for HTTP long-polling and WebSocket transports.
- Support for XHR2 and XHR browsers.
- Support for text and binary messages.
- Support for gzip and deflate HTTP compression.
- Configurable CORS responses, to avoid cross-origin problems with browsers.

CHAPTER 1

What is Socket.IO?

Socket.IO is a transport protocol that enables real-time bidirectional event-based communication between clients (typically web browsers) and a server. The original implementations of the client and server components are written in JavaScript.

CHAPTER 2

Getting Started

The Socket.IO server can be installed with pip:

```
pip install python-socketio
```

The following is a basic example of a Socket.IO server that uses the aiohttp framework for asyncio (Python 3.5+ only):

```
from aiohttp import web
import socketio

sio = socketio.AsyncServer()
app = web.Application()
sio.attach(app)

async def index(request):
    """Serve the client-side application."""
    with open('index.html') as f:
        return web.Response(text=f.read(), content_type='text/html')

@sio.on('connect', namespace='/chat')
def connect(sid, environ):
    print("connect ", sid)

@sio.on('chat message', namespace='/chat')
async def message(sid, data):
    print("message ", data)
    await sio.emit('reply', room=sid)

@sio.on('disconnect', namespace='/chat')
def disconnect(sid):
    print('disconnect ', sid)

app.router.add_static('/static', 'static')
app.router.add_get('/', index)
```

```
if __name__ == '__main__':
    web.run_app(app)
```

And below is a similar example, but using Flask and Eventlet. This example is compatible with Python 2.7 and 3.3+:

```
import socketio
import eventlet
from flask import Flask, render_template

sio = socketio.Server()
app = Flask(__name__)

@app.route('/')
def index():
    """Serve the client-side application."""
    return render_template('index.html')

@sio.on('connect')
def connect(sid, environ):
    print('connect ', sid)

@sio.on('my message')
def message(sid, data):
    print('message ', data)

@sio.on('disconnect')
def disconnect(sid):
    print('disconnect ', sid)

if __name__ == '__main__':
    # wrap Flask application with socketio's middleware
    app = socketio.Middleware(sio, app)

    # deploy as an eventlet WSGI server
    eventlet.wsgi.server(eventlet.listen(('', 8000)), app)
```

The client-side application must include the socket.io-client library (versions 1.3.5 or newer recommended).

Each time a client connects to the server the `connect` event handler is invoked with the `sid` (session ID) assigned to the connection and the WSGI environment dictionary. The server can inspect authentication or other headers to decide if the client is allowed to connect. To reject a client the handler must return `False`.

When the client sends an event to the server, the appropriate event handler is invoked with the `sid` and the message, which can be a single or multiple arguments. The application can define as many events as needed and associate them with event handlers. An event is defined simply by a name.

When a connection with a client is broken, the `disconnect` event is called, allowing the application to perform cleanup.

CHAPTER 3

Server

Socket.IO servers are instances of class *socketio.Server*, which can be combined with a WSGI compliant application using *socketio.Middleware*:

```
# create a Socket.IO server
sio = socketio.Server()

# wrap WSGI application with socketio's middleware
app = socketio.Middleware(sio, app)
```

For asyncio based servers, the *socketio.AsyncServer* class provides a coroutine friendly server:

```
# create a Socket.IO server
sio = socketio.AsyncServer()

# attach server to application
sio.attach(app)
```

Event handlers for servers are register using the *socketio.Server.on()* method:

```
@sio.on('my custom event')
def my_custom_event():
    pass
```

For asyncio servers, event handlers can be regular functions or coroutines:

```
@sio.on('my custom event')
async def my_custom_event():
    await sio.emit('my reply')
```

CHAPTER 4

Rooms

Because Socket.IO is a bidirectional protocol, the server can send messages to any connected client at any time. To make it easy to address groups of clients, the application can put clients into rooms, and then address messages to the entire room.

When clients first connect, they are assigned to their own rooms, named with the session ID (the `sid` argument passed to all event handlers). The application is free to create additional rooms and manage which clients are in them using the *socketio.Server.enter_room()* and *socketio.Server.leave_room()* methods. Clients can be in as many rooms as needed and can be moved between rooms as often as necessary. The individual rooms assigned to clients when they connect are not special in any way, the application is free to add or remove clients from them, though once it does that it will lose the ability to address individual clients.

```
@sio.on('enter room')
def enter_room(sid, data):
    sio.enter_room(sid, data['room'])

@sio.on('leave room')
def leave_room(sid, data):
    sio.leave_room(sid, data['room'])
```

The *socketio.Server.emit()* method takes an event name, a message payload of type `str`, `bytes`, `list`, `dict` or `tuple`, and the recipient room. When sending a `tuple`, the elements in it need to be of any of the other four allowed types. The elements of the tuple will be passed as multiple arguments to the client-side callback function. To address an individual client, the `sid` of that client should be given as room (assuming the application did not alter these initial rooms). To address all connected clients, the `room` argument should be omitted.

```
@sio.on('my message')
def message(sid, data):
    print('message ', data)
    sio.emit('my reply', data, room='my room')
```

Often when broadcasting a message to group of users in a room, it is desirable that the sender does not receive its own message. The *socketio.Server.emit()* method provides an optional `skip_sid` argument to specify a client that should be skipped during the broadcast.

```
@sio.on('my message')
def message(sid, data):
    print('message ', data)
    sio.emit('my reply', data, room='my room', skip_sid=sid)
```

CHAPTER 5

Responses

When a client sends an event to the server, it can optionally provide a callback function, to be invoked with a response provided by the server. The server can provide a response simply by returning it from the corresponding event handler.

```
@sio.on('my event', namespace='/chat')
def my_event_handler(sid, data):
    # handle the message
    return "OK", 123
```

The event handler can return a single value, or a tuple with several values. The callback function on the client side will be invoked with these returned values as arguments.

CHAPTER 6

Callbacks

The server can also request a response to an event sent to a client. The *socketio.Server.emit()* method has an optional `callback` argument that can be set to a callable. When this argument is given, the callable will be invoked with the arguments returned by the client as a response.

Using callback functions when broadcasting to multiple clients is not recommended, as the callback function will be invoked once for each client that received the message.

CHAPTER 7

Namespaces

The Socket.IO protocol supports multiple logical connections, all multiplexed on the same physical connection. Clients can open multiple connections by specifying a different *namespace* on each. A namespace is given by the client as a pathname following the hostname and port. For example, connecting to *http://example.com:8000/chat* would open a connection to the namespace */chat*.

Each namespace is handled independently from the others, with separate session IDs (`sid``s), event handlers and rooms. It is important that applications that use multiple namespaces specify the correct namespace when setting up their event handlers and rooms, using the optional ``namespace argument available in all the methods in the *socketio.Server* class.

When the `namespace` argument is omitted, set to `None` or to `'/'`, a default namespace is used.

CHAPTER 8

Class-Based Namespaces

As an alternative to the decorator-based event handlers, the event handlers that belong to a namespace can be created as methods of a subclass of *socketio.Namespace*:

```
class MyCustomNamespace(socketio.Namespace):
    def on_connect(self, sid, environ):
        pass

    def on_disconnect(self, sid):
        pass

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

sio.register_namespace(MyCustomNamespace('/test'))
```

For asyncio based severs, namespaces must inherit from *socketio.AsyncNamespace*, and can define event handlers as regular methods or coroutines:

```
class MyCustomNamespace(socketio.AsyncNamespace):
    def on_connect(self, sid, environ):
        pass

    def on_disconnect(self, sid):
        pass

    async def on_my_event(self, sid, data):
        await self.emit('my_response', data)

sio.register_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 used 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 *socketio.Server* and *socketio.AsyncServer* classes that default to the proper namespace when the `namespace` argument is not given.

In the case that an event has a handler in a class-based namespace, and also a decorator-based function handler, only the standalone function handler is invoked.

It is important to note that class-based namespaces are singletons. This means that a single instance of a namespace class is used for all clients, and consequently, a namespace instance cannot be used to store client specific information.

CHAPTER 9

Using a Message Queue

The Socket.IO server owns the socket connections to all the clients, so it is the only process that can emit events to them. Unfortunately this becomes a limitation for many applications that use more than one process. A common need is to emit events to clients from a process other than the server, for example a Celery worker.

To enable these auxiliary processes to emit events, the server can be configured to listen for externally issued events on a message queue such as Redis or RabbitMQ. Processes that need to emit events to client then post these events to the queue.

Another situation in which the use of a message queue is necessary is with high traffic applications that work with large number of clients. To support these clients, it may be necessary to horizontally scale the Socket.IO server by splitting the client list among multiple server processes. In this type of installation, each server processes owns the connections to a subset of the clients. To make broadcasting work in this environment, the servers communicate with each other through the message queue.

Kombu

One of the messaging options offered by this package to access the message queue is Kombu , which means that any message queue supported by this package can be used. Kombu can be installed with pip:

```
pip install kombu
```

To use RabbitMQ or other AMQP protocol compatible queues, that is the only required dependency. But for other message queues, Kombu may require additional packages. For example, to use a Redis queue, Kombu needs the Python package for Redis installed as well:

```
pip install redis
```

The appropriate message queue service, such as RabbitMQ or Redis, must also be installed. To configure a Socket.IO server to connect to a Kombu queue, the `client_manager` argument must be passed in the server creation. The following example instructs the server to connect to a Redis service running on the same host and on the default port:

```
mgr = socketio.KombuManager('redis://')
sio = socketio.Server(client_manager=mgr)
```

For a RabbitMQ queue also running on the local server with default credentials, the configuration is as follows:

```
mgr = socketio.KombuManager('amqp://')
sio = socketio.Server(client_manager=mgr)
```

The URL passed to the `KombuManager` constructor is passed directly to Kombu's Connection object, so the Kombu documentation should be consulted for information on how to connect to the message queue appropriately.

Note that Kombu currently does not support asyncio, so it cannot be used with the *`socketio.AsyncServer`* class.

Redis

To use a Redis message queue, the Python package for Redis must also be installed:

```
# WSGI server
pip install redis

# asyncio server
pip install aioredis
```

Native Redis support is accessed through the *`socketio.RedisManager`* and *`socketio.AsyncRedisManager`* classes. These classes connect directly to the Redis store and use the queue's pub/sub functionality:

```
# WSGI server
mgr = socketio.RedisManager('redis://')
sio = socketio.Server(client_manager=mgr)

# asyncio server
mgr = socketio.AsyncRedisManager('redis://')
sio = socketio.AsyncServer(client_manager=mgr)
```

Horizontal scaling

If multiple Socket.IO servers are connected to the same message queue, they automatically communicate with each other and manage a combined client list, without any need for additional configuration. When a load balancer such as nginx is used, this provides virtually unlimited scaling capabilities for the server.

Emitting from external processes

To have a process other than a server connect to the queue to emit a message, the same client manager classes can be used as standalone objects. In this case, the `write_only` argument should be set to `True` to disable the creation of a listening thread, which only makes sense in a server. For example:

```
# connect to the redis queue through Kombu
external_sio = socketio.KombuManager('redis://', write_only=True)
```

```
# emit an event
external_sio.emit('my event', data={'foo': 'bar'}, room='my room')
```

CHAPTER 10

Deployment

The following sections describe a variety of deployment strategies for Socket.IO servers.

Sanic

Sanic is a very efficient asynchronous web server for Python 3.5 and newer.

Instances of class `socketio.AsyncServer` will automatically use Sanic for asynchronous operations if the framework is installed. To request its use explicitly, the `async_mode` option can be given in the constructor:

```
sio = socketio.AsyncServer(async_mode='sanic')
```

A server configured for aiohttp must be attached to an existing application:

```
app = web.Application()
sio.attach(app)
```

The Sanic application can define regular routes that will coexist with the Socket.IO server. A typical pattern is to add routes that serve a client application and any associated static files.

The Sanic application is then executed in the usual manner:

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

aiohttp

Aiohttp is a framework with support for HTTP and WebSocket, based on asyncio. Support for this framework is limited to Python 3.5 and newer.

Instances of class `socketio.AsyncServer` will automatically use aiohttp for asynchronous operations if the library is installed. To request its use explicitly, the `async_mode` option can be given in the constructor:

```
sio = socketio.AsyncServer(async_mode='aiohttp')
```

A server configured for aiohttp must be attached to an existing application:

```
app = web.Application()
sio.attach(app)
```

The aiohttp application can define regular routes that will coexist with the Socket.IO server. A typical pattern is to add routes that serve a client application and any associated static files.

The aiohttp application is then executed in the usual manner:

```
if __name__ == '__main__':
    web.run_app(app)
```

Eventlet

Eventlet is a high performance concurrent networking library for Python 2 and 3 that uses coroutines, enabling code to be written in the same style used with the blocking standard library functions. An Socket.IO server deployed with eventlet has access to the long-polling and WebSocket transports.

Instances of class `socketio.Server` will automatically use eventlet for asynchronous operations if the library is installed. To request its use explicitly, the `async_mode` option can be given in the constructor:

```
sio = socketio.Server(async_mode='eventlet')
```

A server configured for eventlet is deployed as a regular WSGI application, using the provided `socketio.Middleware`:

```
app = socketio.Middleware(sio)
import eventlet
eventlet.wsgi.server(eventlet.listen(('', 8000)), app)
```

An alternative to running the eventlet WSGI server as above is to use gunicorn, a fully featured pure Python web server. The command to launch the application under gunicorn is shown below:

```
$ gunicorn -k eventlet -w 1 module:app
```

Due to limitations in its load balancing algorithm, gunicorn can only be used with one worker process, so the `-w` option cannot be set to a value higher than 1. A single eventlet worker can handle a large number of concurrent clients, each handled by a greenlet.

Eventlet provides a `monkey_patch()` function that replaces all the blocking functions in the standard library with equivalent asynchronous versions. While python-socketio does not require monkey patching, other libraries such as database drivers are likely to require it.

Gevent

Gevent is another asynchronous framework based on coroutines, very similar to eventlet. An Socket.IO server deployed with gevent has access to the long-polling transport. If project gevent-websocket is installed, the WebSocket transport is also available.

Instances of class `socketio.Server` will automatically use gevent for asynchronous operations if the library is installed and eventlet is not installed. To request gevent to be selected explicitly, the `async_mode` option can be given in the constructor:

```
sio = socketio.Server(async_mode='gevent')
```

A server configured for gevent is deployed as a regular WSGI application, using the provided `socketio.Middleware`:

```
app = socketio.Middleware(sio)
from gevent import pywsgi
pywsgi.WSGIServer(('', 8000), app).serve_forever()
```

If the WebSocket transport is installed, then the server must be started as follows:

```
from gevent import pywsgi
from geventwebsocket.handler import WebSocketHandler
app = socketio.Middleware(sio)
pywsgi.WSGIServer(('', 8000), app,
                  handler_class=WebSocketHandler).serve_forever()
```

An alternative to running the gevent WSGI server as above is to use gunicorn, a fully featured pure Python web server. The command to launch the application under gunicorn is shown below:

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

Or to include WebSocket:

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

Same as with eventlet, due to limitations in its load balancing algorithm, gunicorn can only be used with one worker process, so the `-w` option cannot be higher than 1. A single gevent worker can handle a large number of concurrent clients through the use of greenlets.

Gevent provides a `monkey_patch()` function that replaces all the blocking functions in the standard library with equivalent asynchronous versions. While python-socketio does not require monkey patching, other libraries such as database drivers are likely to require it.

Gevent with uWSGI

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

Instances of class `socketio.Server` will automatically use this option for asynchronous operations if both gevent and uWSGI are installed and eventlet is not installed. To request this asynchoronous mode explicitly, the `async_mode` option can be given in the constructor:

```
# gevent with uWSGI
sio = socketio.Server(async_mode='gevent_uwsgi')
```

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 `latency.py` example on port 5000:

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

Standard Threading Library

While not comparable to eventlet and gevent in terms of performance, the Socket.IO server can also be configured to work with multi-threaded web servers that use standard Python threads. This is an ideal setup to use with development servers such as Werkzeug. Only the long-polling transport is currently available when using standard threads.

Instances of class `socketio.Server` will automatically use the threading mode if neither eventlet nor gevent are not installed. To request the threading mode explicitly, the `async_mode` option can be given in the constructor:

```
sio = socketio.Server(async_mode='threading')
```

A server configured for threading is deployed as a regular web application, using any WSGI complaint multi-threaded server. The example below deploys an Socket.IO application combined with a Flask web application, using Flask's development web server based on Werkzeug:

```
sio = socketio.Server(async_mode='threading')
app = Flask(__name__)
app.wsgi_app = socketio.Middleware(sio, app.wsgi_app)

# ... Socket.IO and Flask handler functions ...

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

When using the threading mode, it is important to ensure that the WSGI server can handle multiple concurrent requests using threads, since a client can have up to two outstanding requests at any given time. The Werkzeug server is single-threaded by default, so the `threaded=True` option is required.

Note that servers that use worker processes instead of threads, such as gunicorn, do not support a Socket.IO server configured in threading mode.

Multi-process deployments

Socket.IO is a stateful protocol, which makes horizontal scaling more difficult. To deploy a cluster of Socket.IO processes (hosted on one or multiple servers), the following conditions must be met:

- Each Socket.IO process must be able to handle multiple requests, either by using asyncio, eventlet, gevent, or standard threads. Worker processes that only handle one request at a time are not supported.
- The load balancer must be configured to always forward requests from a client to the same worker process. Load balancers call this *sticky sessions*, or *session affinity*.
- The worker processes communicate with each other through a message queue, which must be installed and configured. See the section on using message queues above for instructions.

CHAPTER 11

API Reference

class socketio.**Middleware**(*socketio_app*, *wsgi_app=None*, *socketio_path='socket.io'*)

WSGI middleware for Socket.IO.

This middleware dispatches traffic to a Socket.IO application, and optionally forwards regular HTTP traffic to a WSGI application.

Parameters

- **socketio_app** – The Socket.IO server.
- **wsgi_app** – The WSGI app that receives all other traffic.
- **socketio_path** – The endpoint where the Socket.IO application should be installed. The default value is appropriate for most cases.

Example usage:

```
import socketio
import eventlet
from . import wsgi_app

sio = socketio.Server()
app = socketio.Middleware(sio, wsgi_app)
eventlet.wsgi.server(eventlet.listen(('', 8000)), app)
```

class socketio.**Server**(*client_manager=None*, *logger=False*, *binary=False*, *json=None*, *async_handlers=False*, ***kwargs*)

A Socket.IO server.

This class implements a fully compliant Socket.IO web server with support for websocket and long-polling transports.

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.

- **logger** – To enable logging set to `True` or pass a logger object to use. To disable logging set to `False`. The default is `False`.
- **binary** – `True` to support binary payloads, `False` to treat all payloads as text. On Python 2, if this is set to `True`, `unicode` values are treated as text, and `str` and `bytes` values are treated as binary. This option has no effect on Python 3, where text and binary payloads are always automatically discovered.
- **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.
- **async_handlers** – If set to `True`, event handlers are executed in separate threads. To run handlers synchronously, set to `False`. The default is `False`.
- **kwargs** – Connection parameters for the underlying Engine.IO server.

The Engine.IO 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_timeout** – The time in seconds that the client waits for the server to respond before disconnecting. The default is 60 seconds.
- **ping_interval** – The interval in seconds at which the client pings the server. The default is 25 seconds.
- **max_http_buffer_size** – The maximum size of a message when using the polling transport. The default is 100,000,000 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** – Name of the HTTP cookie that contains the client session id. If set to `None`, a cookie is not sent to the client. The default is `'io'`.
- **cors_allowed_origins** – List of origins that are allowed to connect to this server. All origins are allowed by default.
- **cors_credentials** – Whether credentials (cookies, authentication) are allowed in requests to this server. 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`.

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

Close a room.

This function removes all the clients from the given room.

Parameters

- **room** – Room name.

- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the default namespace is used.

disconnect(*sid*, *namespace=None*)
Disconnect a client.

Parameters

- **sid** – Session ID of the client.
- **namespace** – The Socket.IO namespace to disconnect. If this argument is omitted the default namespace is used.

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

Parameters

- **event** – The event name. It can be any string. The event names `'connect'`, `'message'` and `'disconnect'` are reserved and should not be used.
- **data** – The data to send to the client or clients. Data can be of type `str`, `bytes`, `list` or `dict`. If a `list` or `dict`, the data will be serialized as JSON.
- **room** – The recipient of the message. This can be set to the session ID of a client to address that client's room, or to any custom room created by the application, If this argument is omitted the event is broadcasted to all connected clients.
- **skip_sid** – The session ID of a client to skip when broadcasting to a room or to all clients. This can be used to prevent a message from being sent to the sender.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the event is emitted to the default namespace.
- **callback** – If given, this function will be called to acknowledge the 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.
- **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. It is recommended to always leave this parameter with its default value of `False`.

enter_room(*sid*, *room*, *namespace=None*)
Enter a room.

This function adds the client to a room. The *emit()* and *send()* functions can optionally broadcast events to all the clients in a room.

Parameters

- **sid** – Session ID of the client.
- **room** – Room name. If the room does not exist it is created.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the default namespace is used.

handle_request(*environ*, *start_response*)
Handle an HTTP request from the client.

This is the entry point of the Socket.IO application, using the same interface as a WSGI application. For the typical usage, this function is invoked by the *Middleware* instance, but it can be invoked directly when the middleware is not used.

Parameters

- **environ** – The WSGI environment.
- **start_response** – The WSGI `start_response` function.

This function returns the HTTP response body to deliver to the client as a byte sequence.

leave_room(*sid*, *room*, *namespace=None*)
Leave a room.

This function removes the client from a room.

Parameters

- **sid** – Session ID of the client.
- **room** – Room name.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the default namespace is used.

on(*event*, *handler=None*, *namespace=None*)
Register an event handler.

Parameters

- **event** – The event name. It can be any string. The event names `'connect'`, `'message'` and `'disconnect'` are reserved and should not be used.
- **handler** – The function that should be invoked to handle the event. When this parameter is not given, the method acts as a decorator for the handler function.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the handler is associated with the default namespace.

Example usage:

```
# as a decorator:
@socket_io.on('connect', namespace='/chat')
def connect_handler(sid, environ):
    print('Connection request')
    if environ['REMOTE_ADDR'] in blacklisted:
        return False  # reject

# as a method:
def message_handler(sid, msg):
    print('Received message: ', msg)
    eio.send(sid, 'response')
socket_io.on('message', namespace='/chat', message_handler)
```

The handler function receives the `sid` (session ID) for the client as first argument. The `'connect'` event handler receives the WSGI environment as a second argument, and can return `False` to reject the connection. The `'message'` handler and handlers for custom event names receive the message payload as a second argument. Any values returned from a message handler will be passed to the client's acknowledgement callback function if it exists. The `'disconnect'` handler does not take a second argument.

register_namespace(*namespace_handler*)
Register a namespace handler object.

Parameters **namespace_handler** – An instance of a *Namespace* subclass that handles all the event traffic for a namespace.

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

Parameters

- **sid** – Session ID of the client.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the default namespace is used.

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

This function emits an event with the name `'message'`. Use *emit()* to issue custom event names.

Parameters

- **data** – The data to send to the client or clients. Data can be of type `str`, `bytes`, `list` or `dict`. If a `list` or `dict`, the data will be serialized as JSON.
- **room** – The recipient of the message. This can be set to the session ID of a client to address that client's room, or to any custom room created by the application, If this argument is omitted the event is broadcasted to all connected clients.
- **skip_sid** – The session ID of a client to skip when broadcasting to a room or to all clients. This can be used to prevent a message from being sent to the sender.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the event is emitted to the default namespace.
- **callback** – If given, this function will be called to acknowledge the 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.
- **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. It is recommended to always leave this parameter with its default value of `False`.

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 compatible with the *Thread* class in the Python standard library. The *start()* method on this object is already called by this function.

transport(*sid*)
Return the name of the transport used by the client.

The two possible values returned by this function are `'polling'` and `'websocket'`.

Parameters **sid** – The session of the client.

class socketio.**AsyncServer**(*client_manager=None*, *logger=False*, *json=None*, *async_handlers=False*, ***kwargs*)

A Socket.IO server for asyncio.

This class implements a fully compliant Socket.IO web server with support for websocket and long-polling transports, compatible with the asyncio framework on Python 3.5 or newer.

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.
- **logger** – To enable logging set to `True` or pass a logger object to use. To disable logging set to `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.
- **async_handlers** – If set to `True`, event handlers are executed in separate threads. To run handlers synchronously, set to `False`. The default is `False`.
- **kwargs** – Connection parameters for the underlying Engine.IO server.

The Engine.IO 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 "aiohttp". If this argument is not given, an async mode is chosen based on the installed packages.
- **ping_timeout** – The time in seconds that the client waits for the server to respond before disconnecting.
- **ping_interval** – The interval in seconds at which the client pings the server.
- **max_http_buffer_size** – The maximum size of a message when using the polling transport.
- **allow_upgrades** – Whether to allow transport upgrades or not.
- **http_compression** – Whether to compress packages when using the polling transport.
- **compression_threshold** – Only compress messages when their byte size is greater than this value.
- **cookie** – Name of the HTTP cookie that contains the client session id. If set to `None`, a cookie is not sent to the client.
- **cors_allowed_origins** – List of origins that are allowed to connect to this server. All origins are allowed by default.
- **cors_credentials** – Whether credentials (cookies, authentication) are allowed in requests to this server.
- **engineio_logger** – To enable Engine.IO logging set to `True` or pass a logger object to use. To disable logging set to `False`.

attach(*app*, *socketio_path='socket.io'*)

Attach the Socket.IO server to an application.

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

This function removes all the clients from the given room.

Parameters

- **room** – Room name.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the default namespace is used.

Note: this method is a coroutine.

disconnect(*sid*, *namespace=None*)
Disconnect a client.

Parameters

- **sid** – Session ID of the client.
- **namespace** – The Socket.IO namespace to disconnect. If this argument is omitted the default namespace is used.

Note: this method is a coroutine.

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

Parameters

- **event** – The event name. It can be any string. The event names `'connect'`, `'message'` and `'disconnect'` are reserved and should not be used.
- **data** – The data to send to the client or clients. Data can be of type `str`, `bytes`, `list` or `dict`. If a `list` or `dict`, the data will be serialized as JSON.
- **room** – The recipient of the message. This can be set to the session ID of a client to address that client's room, or to any custom room created by the application, If this argument is omitted the event is broadcasted to all connected clients.
- **skip_sid** – The session ID of a client to skip when broadcasting to a room or to all clients. This can be used to prevent a message from being sent to the sender.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the event is emitted to the default namespace.
- **callback** – If given, this function will be called to acknowledge the 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.
- **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. It is recommended to always leave this parameter with its default value of `False`.

Note: this method is a coroutine.

enter_room(*sid*, *room*, *namespace=None*)
Enter a room.

This function adds the client to a room. The *emit()* and *send()* functions can optionally broadcast events to all the clients in a room.

Parameters

- **sid** – Session ID of the client.
- **room** – Room name. If the room does not exist it is created.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the default namespace is used.

handle_request(**args*, ***kwargs*)
Handle an HTTP request from the client.

This is the entry point of the Socket.IO application. This function returns the HTTP response body to deliver to the client.

Note: this method is a coroutine.

leave_room(*sid*, *room*, *namespace=None*)
Leave a room.

This function removes the client from a room.

Parameters

- **sid** – Session ID of the client.
- **room** – Room name.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the default namespace is used.

on(*event*, *handler=None*, *namespace=None*)
Register an event handler.

Parameters

- **event** – The event name. It can be any string. The event names `'connect'`, `'message'` and `'disconnect'` are reserved and should not be used.
- **handler** – The function that should be invoked to handle the event. When this parameter is not given, the method acts as a decorator for the handler function.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the handler is associated with the default namespace.

Example usage:

```
# as a decorator:
@socket_io.on('connect', namespace='/chat')
def connect_handler(sid, environ):
    print('Connection request')
    if environ['REMOTE_ADDR'] in blacklisted:
        return False  # reject

# as a method:
def message_handler(sid, msg):
    print('Received message: ', msg)
    eio.send(sid, 'response')
socket_io.on('message', namespace='/chat', message_handler)
```

The handler function receives the `sid` (session ID) for the client as first argument. The `'connect'` event handler receives the WSGI environment as a second argument, and can return `False` to reject the connection. The `'message'` handler and handlers for custom event names receive the message payload as a second argument. Any values returned from a message handler will be passed to the client's acknowledgement callback function if it exists. The `'disconnect'` handler does not take a second argument.

register_namespace(*namespace_handler*)
Register a namespace handler object.

Parameters **namespace_handler** – An instance of a *Namespace* subclass that handles all the event traffic for a namespace.

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

Parameters

- **sid** – Session ID of the client.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the default namespace is used.

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

This function emits an event with the name `'message'`. Use *emit()* to issue custom event names.

Parameters

- **data** – The data to send to the client or clients. Data can be of type `str`, `bytes`, `list` or `dict`. If a `list` or `dict`, the data will be serialized as JSON.
- **room** – The recipient of the message. This can be set to the session ID of a client to address that client's room, or to any custom room created by the application, If this argument is omitted the event is broadcasted to all connected clients.
- **skip_sid** – The session ID of a client to skip when broadcasting to a room or to all clients. This can be used to prevent a message from being sent to the sender.
- **namespace** – The Socket.IO namespace for the event. If this argument is omitted the event is emitted to the default namespace.
- **callback** – If given, this function will be called to acknowledge the 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.
- **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. It is recommended to always leave this parameter with its default value of `False`.

Note: this method is a coroutine.

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.

Note: this method is a coroutine.

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. Must be a coroutine.

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

The return value is a `asyncio.Task` object.

Note: this method is a coroutine.

transport(*sid*)
Return the name of the transport used by the client.

The two possible values returned by this function are `'polling'` and `'websocket'`.

Parameters **sid** – The session of the client.

class `socketio.`**Namespace**(*namespace=None*)
Base class for class-based namespaces.

A class-based namespace is a class that contains all the event handlers for a Socket.IO namespace. The event handlers are methods of the class with the prefix `on_`, such as `on_connect`, `on_disconnect`, `on_message`, `on_json`, and so on.

Parameters **namespace** – The Socket.IO namespace to be used with all the event handlers defined in this class. If this argument is omitted, the default namespace is used.

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

The only difference with the *socketio.Server.close_room()* method is that when the `namespace` argument is not given the namespace associated with the class is used.

disconnect(*sid*, *namespace=None*)
Disconnect a client.

The only difference with the *socketio.Server.disconnect()* method is that when the `namespace` argument is not given the namespace associated with the class is used.

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

The only difference with the *socketio.Server.emit()* method is that when the `namespace` argument is not given the namespace associated with the class is used.

enter_room(*sid*, *room*, *namespace=None*)
Enter a room.

The only difference with the *socketio.Server.enter_room()* method is that when the `namespace` argument is not given the namespace associated with the class is used.

leave_room(*sid*, *room*, *namespace=None*)
Leave a room.

The only difference with the *socketio.Server.leave_room()* method is that when the `namespace` argument is not given the namespace associated with the class is used.

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

The only difference with the *socketio.Server.rooms()* method is that when the `namespace` argument is not given the namespace associated with the class is used.

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

The only difference with the *socketio.Server.send()* method is that when the namespace argument is not given the namespace associated with the class is used.

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 overriden if special dispatching rules are needed, or if having a single method that catches all events is desired.

class socketio.**AsyncNamespace**(*namespace=None*)
Base class for asyncio class-based namespaces.

A class-based namespace is a class that contains all the event handlers for a Socket.IO namespace. The event handlers are methods of the class with the prefix on_, such as on_connect, on_disconnect, on_message, on_json, and so on. These can be regular functions or coroutines.

Parameters **namespace** – The Socket.IO namespace to be used with all the event handlers defined in this class. If this argument is omitted, the default namespace is used.

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

The only difference with the *socketio.Server.close_room()* method is that when the namespace argument is not given the namespace associated with the class is used.

Note: this method is a coroutine.

disconnect(*sid*, *namespace=None*)
Disconnect a client.

The only difference with the *socketio.Server.disconnect()* method is that when the namespace argument is not given the namespace associated with the class is used.

Note: this method is a coroutine.

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

The only difference with the *socketio.Server.emit()* method is that when the namespace argument is not given the namespace associated with the class is used.

Note: this method is a coroutine.

enter_room(*sid*, *room*, *namespace=None*)
Enter a room.

The only difference with the *socketio.Server.enter_room()* method is that when the namespace argument is not given the namespace associated with the class is used.

leave_room(*sid*, *room*, *namespace=None*)
Leave a room.

The only difference with the *socketio.Server.leave_room()* method is that when the namespace argument is not given the namespace associated with the class is used.

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

The only difference with the *socketio.Server.rooms()* method is that when the namespace argument is not given the namespace associated with the class is used.

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

The only difference with the *socketio.Server.send()* method is that when the `namespace` argument is not given the namespace associated with the class is used.

Note: this method is a coroutine.

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 overriden if special dispatching rules are needed, or if having a single method that catches all events is desired.

Note: this method is a coroutine.

class socketio.**BaseManager**
Manage client connections.

This class keeps track of all the clients and the rooms they are in, to support the broadcasting of messages. The data used by this class is stored in a memory structure, making it appropriate only for single process services. More sophisticated storage backends can be implemented by subclasses.

close_room(*room*, *namespace*)
Remove all participants from a room.

connect(*sid*, *namespace*)
Register a client connection to a namespace.

disconnect(*sid*, *namespace*)
Register a client disconnect from a namespace.

emit(*event*, *data*, *namespace*, *room=None*, *skip_sid=None*, *callback=None*, ***kwargs*)
Emit a message to a single client, a room, or all the clients connected to the namespace.

enter_room(*sid*, *namespace*, *room*)
Add a client to a room.

get_namespaces()
Return an iterable with the active namespace names.

get_participants(*namespace*, *room*)
Return an iterable with the active participants in a room.

get_rooms(*sid*, *namespace*)
Return the rooms a client is in.

initialize()
Invoked before the first request is received. Subclasses can add their initialization code here.

leave_room(*sid*, *namespace*, *room*)
Remove a client from a room.

pre_disconnect(*sid*, *namespace*)
Put the client in the to-be-disconnected list.

This allows the client data structures to be present while the disconnect handler is invoked, but still recognize the fact that the client is soon going away.

trigger_callback(*sid*, *namespace*, *id*, *data*)
Invoke an application callback.

class socketio.**PubSubManager**(*channel='socketio'*, *write_only=False*)
Manage a client list attached to a pub/sub backend.

This is a base class that enables multiple servers to share the list of clients, with the servers communicating events through a pub/sub backend. The use of a pub/sub backend also allows any client connected to the backend to emit events addressed to Socket.IO clients.

The actual backends must be implemented by subclasses, this class only provides a pub/sub generic framework.

Parameters **channel** – The channel name on which the server sends and receives notifications.

emit(*event*, *data*, *namespace=None*, *room=None*, *skip_sid=None*, *callback=None*, ***kwargs*)
Emit a message to a single client, a room, or all the clients connected to the namespace.

This method takes care or propagating the message to all the servers that are connected through the message queue.

The parameters are the same as in *Server.emit()*.

class socketio.**KombuManager**(*url='amqp://guest:guest@localhost:5672//'*, *channel='socketio'*, *write_only=False*)
Client manager that uses kombu for inter-process messaging.

This class implements a client manager backend for event sharing across multiple processes, using RabbitMQ, Redis or any other messaging mechanism supported by kombu.

To use a kombu backend, initialize the *Server* instance as follows:

```
url = 'amqp://user:password@hostname:port//'
server = socketio.Server(client_manager=socketio.KombuManager(url))
```

Parameters

- **url** – The connection URL for the backend messaging queue. Example connection URLs are `'amqp://guest:guest@localhost:5672//'` and `'redis://localhost:6379/'` for RabbitMQ and Redis respectively. Consult the kombu documentation for more on how to construct connection URLs.
- **channel** – The channel name on which the server sends and receives notifications. Must be the same in all the servers.
- **write_only** – If set ot `True`, only initialize to emit events. The default of `False` initializes the class for emitting and receiving.

class socketio.**RedisManager**(*url='redis://localhost:6379/0'*, *channel='socketio'*, *write_only=False*)
Redis based client manager.

This class implements a Redis backend for event sharing across multiple processes. Only kept here as one more example of how to build a custom backend, since the kombu backend is perfectly adequate to support a Redis message queue.

To use a Redis backend, initialize the *Server* instance as follows:

```
url = 'redis://hostname:port/0'
server = socketio.Server(client_manager=socketio.RedisManager(url))
```

Parameters

- **url** – The connection URL for the Redis server. For a default Redis store running on the same host, use `redis://`.
- **channel** – The channel name on which the server sends and receives notifications. Must be the same in all the servers.

- **write_only** – If set ot `True`, only initialize to emit events. The default of `False` initializes the class for emitting and receiving.

class socketio.**AsyncManager**

Manage a client list for an asyncio server.

close_room(*room*, *namespace*)

Remove all participants from a room.

Note: this method is a coroutine.

connect(*sid*, *namespace*)

Register a client connection to a namespace.

disconnect(*sid*, *namespace*)

Register a client disconnect from a namespace.

emit(*event*, *data*, *namespace*, *room=None*, *skip_sid=None*, *callback=None*, ***kwargs*)

Emit a message to a single client, a room, or all the clients connected to the namespace.

Note: this method is a coroutine.

enter_room(*sid*, *namespace*, *room*)

Add a client to a room.

get_namespaces()

Return an iterable with the active namespace names.

get_participants(*namespace*, *room*)

Return an iterable with the active participants in a room.

get_rooms(*sid*, *namespace*)

Return the rooms a client is in.

initialize()

Invoked before the first request is received. Subclasses can add their initialization code here.

leave_room(*sid*, *namespace*, *room*)

Remove a client from a room.

pre_disconnect(*sid*, *namespace*)

Put the client in the to-be-disconnected list.

This allows the client data structures to be present while the disconnect handler is invoked, but still recognize the fact that the client is soon going away.

trigger_callback(*sid*, *namespace*, *id*, *data*)

Invoke an application callback.

Note: this method is a coroutine.

class socketio.**AsyncRedisManager**(*url='redis://localhost:6379/0'*, *channel='socketio'*, *write_only=False*)

Redis based client manager for asyncio servers.

This class implements a Redis backend for event sharing across multiple processes. Only kept here as one more example of how to build a custom backend, since the kombu backend is perfectly adequate to support a Redis message queue.

To use a Redis backend, initialize the *Server* instance as follows:

```
server = socketio.Server(client_manager=socketio.AsyncRedisManager(
    'redis://hostname:port/0'))
```

Parameters

- **url** – The connection URL for the Redis server. For a default Redis store running on the same host, use `redis://`.
- **channel** – The channel name on which the server sends and receives notifications. Must be the same in all the servers.
- **write_only** – If set ot `True`, only initialize to emit events. The default of `False` initializes the class for emitting and receiving.

Python Module Index

S

Index

P

R

S

T