python-socketio Documentation

Miguel Grinberg

Mar 26, 2024

CONTENTS

This projects implements Socket.IO clients and servers that can run standalone or integrated with a variety of Python web frameworks.

GETTING STARTED

1.1 What is Socket.IO?

Socket.IO is a transport protocol that enables real-time bidirectional event-based communication between clients (typically, though not always, web browsers) and a server. The official implementations of the client and server components are written in JavaScript. This package provides Python implementations of both, each with standard and asyncio variants.

1.2 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.

If you are using the Python client and server, the easiest way to ensure compatibility is to use the same version of this package for the client and the server. If you are using this package with a different client or server, then you must ensure the versions are compatible.

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	python-socketio version	python-engineio version
0.9.x	1, 2	1, 2	Not supported	Not supported
1.x and 2.x	3, 4	3	4.x	3.x
3.x and 4.x	5	4	5.x	4.x

1.3 Client Examples

The example that follows shows a simple Python client:

```python
import socketio

sio = socketio.Client()

@sio.event
def connect():
```

```python
    print('connection established')

@sio.event
def my_message(data):
    print('message received with ', data)
    sio.emit('my response', {'response': 'my response'})

@sio.event
def disconnect():
    print('disconnected from server')

sio.connect('http://localhost:5000')
sio.wait()
```

Below is a similar client, coded for `asyncio` (Python 3.5+ only):

```python
import asyncio
import socketio

sio = socketio.AsyncClient()

@sio.event
async def connect():
    print('connection established')

@sio.event
async def my_message(data):
    print('message received with ', data)
    await sio.emit('my response', {'response': 'my response'})

@sio.event
async def disconnect():
    print('disconnected from server')

async def main():
    await sio.connect('http://localhost:5000')
    await sio.wait()

if __name__ == '__main__':
    asyncio.run(main())
```

1.4 Client Features

- Can connect to other Socket.IO servers that are compatible with the JavaScript Socket.IO 1.x and 2.x releases. Work to support release 3.x is in progress.

- Compatible with Python 3.6+.

- Two versions of the client, one for standard Python and another for asyncio.

- Uses an event-based architecture implemented with decorators that hides the details of the protocol.

- Implements HTTP long-polling and WebSocket transports.

- Automatically reconnects to the server if the connection is dropped.

1.5 Server Examples

The following application is a basic server example that uses the Eventlet asynchronous server:

```python
import eventlet
import socketio

sio = socketio.Server()
app = socketio.WSGIApp(sio, static_files={
    '/': {'content_type': 'text/html', 'filename': 'index.html'}
})

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

@sio.event
def my_message(sid, data):
    print('message ', data)

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

if __name__ == '__main__':
    eventlet.wsgi.server(eventlet.listen(('', 5000)), app)
```

Below is a similar application, coded for `asyncio` (Python 3.5+ only) and the Uvicorn web server:

```python
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.event
def connect(sid, environ):
    print("connect ", sid)

@sio.event
async def chat_message(sid, data):
    print("message ", data)

@sio.event
```

```python
def disconnect(sid):
    print('disconnect ', sid)


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


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

1.6 Server Features

- Can connect to servers running other Socket.IO clients that are compatible with the JavaScript client versions 1.x and 2.x. Work to support the 3.x release is in progress.

- Compatible with Python 3.6+.

- Two versions of the server, one for standard Python and another for asyncio.

- Supports large number of clients even on modest hardware due to being asynchronous.

- Can be hosted on any WSGI or ASGI web server including Gunicorn, Uvicorn, eventlet and gevent.

- Can be integrated with WSGI applications written in frameworks such as Flask, Django, etc.

- Can be integrated with aiohttp, sanic and tornado `asyncio` 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.

THE SOCKET.IO CLIENTS

This package contains two Socket.IO clients:

- a "simple" client, which provides a straightforward API that is sufficient for most applications
- an "event-driven" client, which provides access to all the features of the Socket.IO protocol

Each of these clients comes in two variants: one for the standard Python library, and another for asynchronous applications built with the `asyncio` package.

2.1 Installation

To install the standard Python client along with its dependencies, use the following command:

```
pip install "python-socketio[client]"
```

If instead you plan on using the `asyncio` client, then use this:

```
pip install "python-socketio[asyncio_client]"
```

2.2 Using the Simple Client

The advantage of the simple client is that it abstracts away the logic required to maintain a Socket.IO connection. This client handles disconnections and reconnections in a completely transparent way, without adding any complexity to the application.

2.2.1 Creating a Client Instance

The easiest way to create a Socket.IO client is to use the context manager interface:

```python
import socketio

# standard Python
with socketio.SimpleClient() as sio:
    # ... connect to a server and use the client
    # ... no need to manually disconnect!

# asyncio
async with socketio.AsyncSimpleClient() as sio:
```

```
    # ... connect to a server and use the client
    # ... no need to manually disconnect!
```

With this usage the context manager will ensure that the client is properly disconnected before exiting the `with` or `async with` block.

If preferred, a client can be manually instantiated:

```python
import socketio

# standard Python
sio = socketio.SimpleClient()

# asyncio
sio = socketio.AsyncSimpleClient()
```

2.2.2 Connecting to a Server

The connection to a server is established by calling the `connect()` method:

```python
sio.connect('http://localhost:5000')
```

In the case of the `asyncio` client, the method is a coroutine:

```python
await sio.connect('http://localhost:5000')
```

By default the client first connects to the server using the long-polling transport, and then attempts to upgrade the connection to use WebSocket. To connect directly using WebSocket, use the `transports` argument:

```python
sio.connect('http://localhost:5000', transports=['websocket'])
```

Upon connection, the server assigns the client a unique session identifier. The application can find this identifier in the `sid` attribute:

```python
print('my sid is', sio.sid)
```

The Socket.IO transport that is used in the connection can be obtained from the `transport` attribute:

```python
print('my transport is', sio.transport)
```

The transport is given as a string, and can be either `'websocket'` or `'polling'`.

TLS/SSL Support

The client supports TLS/SSL connections. To enable it, use a `https://` connection URL:

```python
sio.connect('https://example.com')
```

Or when using `asyncio`:

```python
await sio.connect('https://example.com')
```

The client verifies server certificates by default. Consult the documentation for the event-driven client for information on how to customize this behavior.

2.2.3 Emitting Events

The client can emit an event to the server using the `emit()` method:

```
sio.emit('my message', {'foo': 'bar'})
```

Or in the case of `asyncio`, as a coroutine:

```
await sio.emit('my message', {'foo': 'bar'})
```

The arguments provided to the method are the name of the event to emit and the optional data that is passed on to the server. The data can be of type `str`, `bytes`, `dict`, `list` or `tuple`. When sending a `list` or a `tuple`, the elements in it need to be of any allowed types except `tuple`. When a tuple is used, the elements of the tuple will be passed as individual arguments to the server-side event handler function.

2.2.4 Receiving Events

The client can wait for the server to emit an event with the `receive()` method:

```
event = sio.receive()
print(f'received event: "{event[0]}" with arguments {event[1:]}')
```

When using `asyncio`, this method needs to be awaited:

```
event = await sio.receive()
print(f'received event: "{event[0]}" with arguments {event[1:]}')
```

The return value of `receive()` is a list. The first element of this list is the event name, while the remaining elements are the arguments passed by the server.

With the usage shown above, the `receive()` method will return only when an event is received from the server. An optional timeout in seconds can be passed to prevent the client from waiting forever:

```
from socketio.exceptions import TimeoutError

try:
    event = sio.receive(timeout=5)
except TimeoutError:
    print('timed out waiting for event')
else:
    print('received event:', event)
```

Or with `asyncio`:

```
from socketio.exceptions import TimeoutError

try:
    event = await sio.receive(timeout=5)
except TimeoutError:
    print('timed out waiting for event')
```

```
else:
    print('received event:', event)
```

2.2.5 Disconnecting from the Server

At any time the client can request to be disconnected from the server by invoking the `disconnect()` method:

```
sio.disconnect()
```

For the `asyncio` client this is a coroutine:

```
await sio.disconnect()
```

2.2.6 Debugging and Troubleshooting

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

```
import socketio

# standard Python
sio = socketio.Client(logger=True, engineio_logger=True)

# asyncio
sio = socketio.AsyncClient(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, unexpected disconnections and other issues.

2.3 Using the Event-Driven Client

2.3.1 Creating a Client Instance

To instantiate an Socket.IO client, simply create an instance of the appropriate client class:

```
import socketio

# standard Python
sio = socketio.Client()

# asyncio
sio = socketio.AsyncClient()
```

2.3.2 Defining Event Handlers

The Socket.IO protocol is event based. When a server wants to communicate with a client it *emits* an event. Each event has a name, and a list of arguments. The client registers event handler functions with the *socketio.Client.event()* or *socketio.Client.on()* decorators:

```python
@sio.event
def message(data):
    print('I received a message!')


@sio.on('my message')
def on_message(data):
    print('I received a message!')
```

In the first example the event name is obtained from the name of the handler function. The second example is slightly more verbose, but it allows the event name to be different than the function name or to include characters that are illegal in function names, such as spaces.

For the `asyncio` client, event handlers can be regular functions as above, or can also be coroutines:

```python
@sio.event
async def message(data):
    print('I received a message!')
```

If the server includes arguments with an event, those are passed to the handler function as arguments.

2.3.3 Catch-All Event and Namespace Handlers

A "catch-all" event handler is invoked for any events that do not have an event handler. You can define a catch-all handler using `'*'` as event name:

```python
@sio.on('*')
def any_event(event, sid, data):
        pass
```

Asyncio servers can also use a coroutine:

```python
@sio.on('*')
async def any_event(event, sid, data):
    pass
```

A catch-all event handler receives the event name as a first argument. The remaining arguments are the same as for a regular event handler.

The `connect` and `disconnect` events have to be defined explicitly and are not invoked on a catch-all event handler.

Similarily, a "catch-all" namespace handler is invoked for any connected namespaces that do not have an explicitly defined event handler. As with catch-all events, `'*'` is used in place of a namespace:

```python
@sio.on('my_event', namespace='*')
def my_event_any_namespace(namespace, sid, data):
    pass
```

For these events, the namespace is passed as first argument, followed by the regular arguments of the event.

Lastly, it is also possible to define a "catch-all" handler for all events on all namespaces:

```
@sio.on('*', namespace='*')
def any_event_any_namespace(event, namespace, sid, data):
    pass
```

Event handlers with catch-all events and namespaces receive the event name and the namespace as first and second arguments.

2.3.4 Connect, Connect Error and Disconnect Event Handlers

The `connect`, `connect_error` and `disconnect` events are special; they are invoked automatically when a client connects or disconnects from the server:

```
@sio.event
def connect():
    print("I'm connected!")

@sio.event
def connect_error(data):
    print("The connection failed!")

@sio.event
def disconnect():
    print("I'm disconnected!")
```

The `connect_error` handler is invoked when a connection attempt fails. If the server provides arguments, these are passed on to the handler. The server can use an argument to provide information to the client regarding the connection failure.

The `disconnect` handler is invoked for application initiated disconnects, server initiated disconnects, or accidental disconnects, for example due to networking failures. In the case of an accidental disconnection, the client is going to attempt to reconnect immediately after invoking the disconnect handler. As soon as the connection is re-established the connect handler will be invoked once again.

The `connect`, `connect_error` and `disconnect` events have to be defined explicitly and are not invoked on a catch-all event handler.

2.3.5 Connecting to a Server

The connection to a server is established by calling the `connect()` method:

```
sio.connect('http://localhost:5000')
```

In the case of the `asyncio` client, the method is a coroutine:

```
await sio.connect('http://localhost:5000')
```

Upon connection, the server assigns the client a unique session identifier. The application can find this identifier in the `sid` attribute:

```
print('my sid is', sio.sid)
```

The Socket.IO transport that is used in the connection can be obtained from the `transport` attribute:

```
print('my transport is', sio.transport)
```

The transport is given as a string, and can be either `'websocket'` or `'polling'`.

TLS/SSL Support

The client supports TLS/SSL connections. To enable it, use a `https://` connection URL:

```
sio.connect('https://example.com')
```

Or when using `asyncio`:

```
await sio.connect('https://example.com')
```

The client will verify the server certificate by default. To disable certificate verification, or to use other less common options such as client certificates, the client must be initialized with a custom HTTP session object that is configured with the desired TLS/SSL options.

The following example disables server certificate verification, which can be useful when connecting to a server that uses a self-signed certificate:

```
http_session = requests.Session()
http_session.verify = False
sio = socketio.Client(http_session=http_session)
sio.connect('https://example.com')
```

And when using `asyncio`:

```
connector = aiohttp.TCPConnector(ssl=False)
http_session = aiohttp.ClientSession(connector=connector)
sio = socketio.AsyncClient(http_session=http_session)
await sio.connect('https://example.com')
```

Instead of disabling certificate verification, you can provide a custom certificate authority bundle to verify the certificate against:

```
http_session = requests.Session()
http_session.verify = '/path/to/ca.pem'
sio = socketio.Client(http_session=http_session)
sio.connect('https://example.com')
```

And for `asyncio`:

```
ssl_context = ssl.create_default_context()
ssl_context.load_verify_locations('/path/to/ca.pem')
connector = aiohttp.TCPConnector(ssl=ssl_context)
http_session = aiohttp.ClientSession(connector=connector)
sio = socketio.AsyncClient(http_session=http_session)
await sio.connect('https://example.com')
```

Below you can see how to use a client certificate to authenticate against the server:

```
http_session = requests.Session()
http_session.cert = ('/path/to/client/cert.pem', '/path/to/client/key.pem')
```

```
sio = socketio.Client(http_session=http_session)
sio.connect('https://example.com')
```

And for `asyncio`:

```
ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_context.load_cert_chain('/path/to/client/cert.pem',
                            '/path/to/client/key.pem')
connector = aiohttp.TCPConnector(ssl=ssl_context)
http_session = aiohttp.ClientSession(connector=connector)
sio = socketio.AsyncClient(http_session=http_session)
await sio.connect('https://example.com')
```

2.3.6 Emitting Events

The client can emit an event to the server using the `emit()` method:

```
sio.emit('my message', {'foo': 'bar'})
```

Or in the case of `asyncio`, as a coroutine:

```
await sio.emit('my message', {'foo': 'bar'})
```

The arguments provided to the method are the name of the event to emit and the optional data that is passed on to the server. The data can be of type `str`, `bytes`, `dict`, `list` or `tuple`. When sending a `list` or a `tuple`, the elements in it need to be of any allowed types except `tuple`. When a tuple is used, the elements of the tuple will be passed as individual arguments to the server-side event handler function.

The `emit()` method can be invoked inside an event handler as a response to a server event, or in any other part of the application, including in background tasks.

2.3.7 Event Callbacks

When a server emits an event to a client, it can optionally provide a callback function, to be invoked as a way of acknowledgment that the server has processed the event. While this is entirely managed by the server, the client can provide a list of return values that are to be passed on to the callback function set up by the server. This is achieved simply by returning the desired values from the handler function:

```
@sio.event
def my_event(sid, data):
    # handle the message
    return "OK", 123
```

Likewise, the client can request a callback function to be invoked after the server has processed an event. The `socketio.Server.emit()` method has an optional `callback` argument that can be set to a callable. If this argument is given, the callable will be invoked after the server has processed the event, and any values returned by the server handler will be passed as arguments to this function.

2.3.8 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. Namespaces use a path syntax starting with a forward slash. A list of namespaces can be given by the client in the `connect()` call. For example, this example creates two logical connections, the default one plus a second connection under the `/chat` namespace:

```
sio.connect('http://localhost:5000', namespaces=['/chat'])
```

To define event handlers on a namespace, the `namespace` argument must be added to the corresponding decorator:

```python
@sio.event(namespace='/chat')
def my_custom_event(sid, data):
    pass

@sio.on('connect', namespace='/chat')
def on_connect():
    print("I'm connected to the /chat namespace!")
```

Likewise, the client can emit an event to the server on a namespace by providing its in the `emit()` call:

```python
sio.emit('my message', {'foo': 'bar'}, namespace='/chat')
```

If the `namespaces` argument of the `connect()` call isn't given, any namespaces used in event handlers are automatically connected.

2.3.9 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.ClientNamespace*:

```python
class MyCustomNamespace(socketio.ClientNamespace):
    def on_connect(self):
        pass

    def on_disconnect(self):
        pass

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

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

For asyncio based servers, namespaces must inherit from *socketio.AsyncClientNamespace*, and can define event handlers as coroutines if desired:

```python
class MyCustomNamespace(socketio.AsyncClientNamespace):
    def on_connect(self):
        pass

    def on_disconnect(self):
        pass
```

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

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

A catch-all class-based namespace handler can be defined by passing `'*'` as the namespace during registration:

```
sio.register_namespace(MyCustomNamespace('*'))
```

When class-based namespaces are used, any events received by the client 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 *socketio.Client* and *socketio.AsyncClient* 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.

2.3.10 Disconnecting from the Server

At any time the client can request to be disconnected from the server by invoking the `disconnect()` method:

```
sio.disconnect()
```

For the `asyncio` client this is a coroutine:

```
await sio.disconnect()
```

2.3.11 Managing Background Tasks

When a client connection to the server is established, a few background tasks will be spawned to keep the connection alive and handle incoming events. The application running on the main thread is free to do any work, as this is not going to prevent the functioning of the Socket.IO client.

If the application does not have anything to do in the main thread and just wants to wait until the connection with the server ends, it can call the `wait()` method:

```
sio.wait()
```

Or in the `asyncio` version:

```
await sio.wait()
```

For the convenience of the application, a helper function is provided to start a custom background task:

```
def my_background_task(my_argument):
    # do some background work here!
    pass

task = sio.start_background_task(my_background_task, 123)
```

The arguments passed to this method are the background function and any positional or keyword arguments to invoke the function with.

Here is the `asyncio` version:

```python
async def my_background_task(my_argument):
    # do some background work here!
    pass

task = sio.start_background_task(my_background_task, 123)
```

Note that this function is not a coroutine, since it does not wait for the background function to end. The background function must be a coroutine.

The `sleep()` method is a second convenience function that is provided for the benefit of applications working with background tasks of their own:

```python
sio.sleep(2)
```

Or for `asyncio`:

```python
await sio.sleep(2)
```

The single argument passed to the method is the number of seconds to sleep for.

2.3.12 Debugging and Troubleshooting

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

```python
import socketio

# standard Python
sio = socketio.Client(logger=True, engineio_logger=True)

# asyncio
sio = socketio.AsyncClient(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, unexpected disconnections and other issues.

THE SOCKET.IO SERVER

This package contains two Socket.IO servers:

- The *socketio.Server()* class creates a server compatible with the Python standard library.
- The *socketio.AsyncServer()* class creates a server compatible with the `asyncio` package.

The methods in the two servers are the same, with the only difference that in the `asyncio` server most methods are implemented as coroutines.

3.1 Installation

To install the Socket.IO server along with its dependencies, use the following command:

```
pip install python-socketio
```

In addition to the server, you will need to select an asynchronous framework or server to use along with it. The list of supported packages is covered in the *Deployment Strategies* section.

3.2 Creating a Server Instance

A Socket.IO server is an instance of class *socketio.Server*. This instance can be transformed into a standard WSGI application by wrapping it with the *socketio.WSGIApp* class:

```python
import socketio

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

# wrap with a WSGI application
app = socketio.WSGIApp(sio)
```

For asyncio based servers, the *socketio.AsyncServer* class provides the same functionality, but in a coroutine friendly format. If desired, The *socketio.ASGIApp* class can transform the server into a standard ASGI application:

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

# wrap with ASGI application
app = socketio.ASGIApp(sio)
```

These two wrappers can also act as middlewares, forwarding any traffic that is not intended to the Socket.IO server to another application. This allows Socket.IO servers to integrate easily into existing WSGI or ASGI applications:

```python
from wsgi import app  # a Flask, Django, etc. application
app = socketio.WSGIApp(sio, app)
```

3.3 Serving Static Files

The Socket.IO server can be configured to serve static files to clients. This is particularly useful to deliver HTML, CSS and JavaScript files to clients when this package is used without a companion web framework.

Static files are configured with a Python dictionary in which each key/value pair is a static file mapping rule. In its simplest form, this dictionary has one or more static file URLs as keys, and the corresponding files in the server as values:

```python
static_files = {
    '/': 'latency.html',
    '/static/socket.io.js': 'static/socket.io.js',
    '/static/style.css': 'static/style.css',
}
```

With this example configuration, when the server receives a request for / (the root URL) it will return the contents of the file `latency.html` in the current directory, and will assign a content type based on the file extension, in this case `text/html`.

Files with the `.html`, `.css`, `.js`, `.json`, `.jpg`, `.png`, `.gif` and `.txt` file extensions are automatically recognized and assigned the correct content type. For files with other file extensions or with no file extension, the `application/octet-stream` content type is used as a default.

If desired, an explicit content type for a static file can be given as follows:

```python
static_files = {
    '/': {'filename': 'latency.html', 'content_type': 'text/plain'},
}
```

It is also possible to configure an entire directory in a single rule, so that all the files in it are served as static files:

```python
static_files = {
    '/static': './public',
}
```

In this example any files with URLs starting with `/static` will be served directly from the `public` folder in the current directory, so for example, the URL `/static/index.html` will return local file `./public/index.html` and the URL `/static/css/styles.css` will return local file `./public/css/styles.css`.

If a URL that ends in a / is requested, then a default filename of `index.html` is appended to it. In the previous example, a request for the `/static/` URL would return local file `./public/index.html`. The default filename to serve for slash-ending URLs can be set in the static files dictionary with an empty key:

```python
static_files = {
    '/static': './public',
    '': 'image.gif',
}
```

With this configuration, a request for `/static/` would return local file `./public/image.gif`. A non-standard content type can also be specified if needed:

```
static_files = {
    '/static': './public',
    '': {'filename': 'image.gif', 'content_type': 'text/plain'},
}
```

The static file configuration dictionary is given as the `static_files` argument to the `socketio.WSGIApp` or `socketio.ASGIApp` classes:

```
# for standard WSGI applications
sio = socketio.Server()
app = socketio.WSGIApp(sio, static_files=static_files)

# for asyncio-based ASGI applications
sio = socketio.AsyncServer()
app = socketio.ASGIApp(sio, static_files=static_files)
```

The routing precedence in these two classes is as follows:

- First, the path is checked against the Socket.IO endpoint.

- Next, the path is checked against the static file configuration, if present.

- If the path did not match the Socket.IO endpoint or any static file, control is passed to the secondary application if configured, else a 404 error is returned.

Note: static file serving is intended for development use only, and as such it lacks important features such as caching. Do not use in a production environment.

3.4 Defining Event Handlers

The Socket.IO protocol is event based. When a client wants to communicate with the server it *emits* an event. Each event has a name, and a list of arguments. The server registers event handler functions with the *socketio.Server.event()* or *socketio.Server.on()* decorators:

```
@sio.event
def my_event(sid, data):
    pass

@sio.on('my custom event')
def another_event(sid, data):
    pass
```

In the first example the event name is obtained from the name of the handler function. The second example is slightly more verbose, but it allows the event name to be different than the function name or to include characters that are illegal in function names, such as spaces.

For asyncio servers, event handlers can optionally be given as coroutines:

```
@sio.event
async def my_event(sid, data):
    pass
```

The `sid` argument is the Socket.IO session id, a unique identifier of each client connection. All the events sent by a given client will have the same `sid` value.

3.5 Catch-All Event and Namespace Handlers

A "catch-all" event handler is invoked for any events that do not have an event handler. You can define a catch-all handler using `'*'` as event name:

```python
@sio.on('*')
def any_event(event, sid, data):
    pass
```

Asyncio servers can also use a coroutine:

```python
@sio.on('*')
async def any_event(event, sid, data):
    pass
```

A catch-all event handler receives the event name as a first argument. The remaining arguments are the same as for a regular event handler.

The `connect` and `disconnect` events have to be defined explicitly and are not invoked on a catch-all event handler.

Similarily, a "catch-all" namespace handler is invoked for any connected namespaces that do not have an explicitly defined event handler. As with catch-all events, `'*'` is used in place of a namespace:

```python
@sio.on('my_event', namespace='*')
def my_event_any_namespace(namespace, sid, data):
    pass
```

For these events, the namespace is passed as first argument, followed by the regular arguments of the event.

Lastly, it is also possible to define a "catch-all" handler for all events on all namespaces:

```python
@sio.on('*', namespace='*')
def any_event_any_namespace(event, namespace, sid, data):
    pass
```

Event handlers with catch-all events and namespaces receive the event name and the namespace as first and second arguments.

3.6 Connect and Disconnect Event Handlers

The `connect` and `disconnect` events are special; they are invoked automatically when a client connects or disconnects from the server:

```python
@sio.event
def connect(sid, environ, auth):
    print('connect ', sid)

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

The `connect` event is an ideal place to perform user authentication, and any necessary mapping between user entities in the application and the `sid` that was assigned to the client. The `environ` argument is a dictionary in standard WSGI format containing the request information, including HTTP headers. The `auth` argument contains any authentication details passed by the client, or `None` if the client did not pass anything. After inspecting the request, the connect event handler can return `False` to reject the connection with the client.

Sometimes it is useful to pass data back to the client being rejected. In that case instead of returning `False` *socketio.exceptions.ConnectionRefusedError* can be raised, and all of its arguments will be sent to the client with the rejection message:

```python
@sio.event
def connect(sid, environ):
    raise ConnectionRefusedError('authentication failed')
```

3.7 Emitting Events

Socket.IO is a bidirectional protocol, so at any time the server can send an event to its connected clients. The *socketio.Server.emit()* method is used for this task:

```python
sio.emit('my event', {'data': 'foobar'})
```

Sometimes the server may want to send an event just to a particular client. This can be achieved by adding a `room` argument to the emit call:

```python
sio.emit('my event', {'data': 'foobar'}, room=user_sid)
```

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 event handler function. The `room` argument is used to identify the client that should receive the event, and is set to the `sid` value assigned to that client's connection with the server. When omitted, the event is broadcasted to all connected clients.

3.8 Event Callbacks

When a client sends an event to the server, it can optionally provide a callback function, to be invoked as a way of acknowledgment that the server has processed the event. While this is entirely managed by the client, the server can provide a list of values that are to be passed on to the callback function, simply by returning them from the handler function:

```python
@sio.event
def my_event(sid, data):
    # handle the message
    return "OK", 123
```

Likewise, the server can request a callback function to be invoked after a client has processed an event. The *socketio.Server.emit()* method has an optional `callback` argument that can be set to a callable. If this argument is given, the callable will be invoked after the client has processed the event, and any values returned by the client will be passed as arguments to this function. Using callback functions when broadcasting to multiple clients is currently not supported.

3.9 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:

```python
@sio.event(namespace='/chat')
def my_custom_event(sid, data):
    pass


@sio.on('my custom event', namespace='/chat')
def my_custom_event(sid, data):
    pass
```

When emitting an event, the `namespace` optional argument is used to specify which namespace to send it on. When the `namespace` argument is omitted, the default Socket.IO namespace, which is named `/`, is used.

3.10 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*:

```python
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 servers, namespaces must inherit from *socketio.AsyncNamespace*, and can define event handlers as coroutines if desired:

```python
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'))
```

A catch-all class-based namespace handler can be defined by passing `'*'` as the namespace during registration:

```
sio.register_namespace(MyCustomNamespace('*'))
```

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 *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.

3.11 Rooms

To make it easy for the server to emit events to groups of related clients, the application can put its clients into "rooms", and then address messages to these rooms.

In the previous section the room argument of the socketio.SocketIO.emit() method was used to designate a specific client as the recipient of the event. This is because upon connection, a personal room for each client is created and named with the sid assigned to the connection. The application is then 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.

```python
@sio.event
def begin_chat(sid):
    sio.enter_room(sid, 'chat_users')

@sio.event
def exit_chat(sid):
    sio.leave_room(sid, 'chat_users')
```

In chat applications it is often desired that an event is broadcasted to all the members of the room except one, which is the originator of the event such as a chat message. The *socketio.Server.emit()* method provides an optional skip_sid argument to indicate a client that should be skipped during the broadcast.

```python
@sio.event
def my_message(sid, data):
    sio.emit('my reply', data, room='chat_users', skip_sid=sid)
```

3.12 User Sessions

The server can maintain application-specific information in a user session dedicated to each connected client. Applications can use the user session to write any details about the user that need to be preserved throughout the life of the connection, such as usernames or user ids.

The `save_session()` and `get_session()` methods are used to store and retrieve information in the user session:

```python
@sio.event
def connect(sid, environ):
    username = authenticate_user(environ)
    sio.save_session(sid, {'username': username})

@sio.event
def message(sid, data):
    session = sio.get_session(sid)
    print('message from ', session['username'])
```

For the `asyncio` server, these methods are coroutines:

```python
@sio.event
async def connect(sid, environ):
    username = authenticate_user(environ)
    await sio.save_session(sid, {'username': username})

@sio.event
async def message(sid, data):
    session = await sio.get_session(sid)
    print('message from ', session['username'])
```

The session can also be manipulated with the *session()* context manager:

```python
@sio.event
def connect(sid, environ):
    username = authenticate_user(environ)
    with sio.session(sid) as session:
        session['username'] = username

@sio.event
def message(sid, data):
    with sio.session(sid) as session:
        print('message from ', session['username'])
```

For the `asyncio` server, an asynchronous context manager is used:

```python
@sio.event
async def connect(sid, environ):
    username = authenticate_user(environ)
    async with sio.session(sid) as session:
        session['username'] = username

@sio.event
async def message(sid, data):
```

```
    async with sio.session(sid) as session:
        print('message from ', session['username'])
```

The `get_session()`, `save_session()` and `session()` methods take an optional `namespace` argument. If this argument isn't provided, the session is attached to the default namespace.

Note: the contents of the user session are destroyed when the client disconnects. In particular, user session contents are not preserved when a client reconnects after an unexpected disconnection from the server.

3.13 Using a Message Queue

When working with distributed applications, it is often necessary to access the functionality of the Socket.IO from multiple processes. There are two specific use cases:

- Applications that use work queues such as Celery may need to emit an event to a client once a background job completes. The most convenient place to carry out this task is the worker process that handled this job.

- Highly available applications may want to use horizontal scaling of the Socket.IO server to be able to handle very large number of concurrent clients.

As a solution to the above problems, the Socket.IO server can be configured to connect to a message queue such as Redis or RabbitMQ, to communicate with other related Socket.IO servers or auxiliary workers.

3.13.1 Redis

To use a Redis message queue, a Python Redis client must be installed:

```
# socketio.Server class
pip install redis
```

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

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

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

The `client_manager` argument instructs the server to connect to the given message queue, and to coordinate with other processes connected to the queue.

3.13.2 Kombu

Kombu is a Python package that provides access to RabbitMQ and many other message queues. It 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 via Kombu, the Python package for Redis needs to be installed as well:

```
pip install redis
```

The queue is configured through the `socketio.KombuManager`:

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

The connection 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 build the correct URL for a given message queue.

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

3.13.3 Kafka

Apache Kafka is supported through the kafka-python package:

```
pip install kafka-python
```

Access to Kafka is configured through the `socketio.KafkaManager` class:

```
mgr = socketio.KafkaManager('kafka://')
sio = socketio.Server(client_manager=mgr)
```

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

3.13.4 AioPika

A RabbitMQ message queue is supported in asyncio applications through the AioPika package:: You need to install aio_pika with pip:

```
pip install aio_pika
```

The RabbitMQ queue is configured through the `socketio.AsyncAioPikaManager` class:

```
mgr = socketio.AsyncAioPikaManager('amqp://')
sio = socketio.AsyncServer(client_manager=mgr)
```

3.13.5 Horizontal Scaling

Socket.IO is a stateful protocol, which makes horizontal scaling more difficult. When deploying a cluster of Socket.IO processes, all processes must connect to the message queue by passing the `client_manager` argument to the server instance. This enables the workers to communicate and coordinate complex operations such as broadcasts.

If the long-polling transport is used, then there are two additional requirements that must be met:

- Each Socket.IO process must be able to handle multiple requests concurrently. This is needed because long-polling clients send two requests in parallel. Worker processes that can 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, so that all requests coming from a client are handled by the same node. Load balancers call this *sticky sessions*, or *session affinity*.

3.13.6 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 as an external process
external_sio = socketio.RedisManager('redis://', write_only=True)

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

A limitation of the write-only client manager object is that it cannot receive callbacks when emitting. When the external process needs to receive callbacks, using a client to connect to the server with read and write support is a better option than a write-only client manager.

3.14 Monitoring and Administration

The Socket.IO server can be configured to accept connections from the official Socket.IO Admin UI. This tool provides real-time information about currently connected clients, rooms in use and events being emitted. It also allows an administrator to manually emit events, change room assignments and disconnect clients. The hosted version of this tool is available at https://admin.socket.io.

Given that enabling this feature can affect the performance of the server, it is disabled by default. To enable it, call the `instrument()` method. For example:

```
import os
import socketio

sio = socketio.Server(cors_allowed_origins=[
    'http://localhost:5000',
    'https://admin.socket.io',
])
sio.instrument(auth={
    'username': 'admin',
    'password': os.environ['ADMIN_PASSWORD'],
})
```

This configures the server to accept connections from the hosted Admin UI client. Administrators can then open https://admin.socket.io in their web browsers and log in with username `admin` and the password given by the `ADMIN_PASSWORD` environment variable. To ensure the Admin UI front end is allowed to connect, CORS is also configured.

Consult the reference documentation to learn about additional configuration options that are available.

3.15 Debugging and Troubleshooting

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

```python
import socketio

# standard Python
sio = socketio.Server(logger=True, engineio_logger=True)

# asyncio
sio = socketio.AsyncServer(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.

3.16 Deployment Strategies

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

3.16.1 Uvicorn, Daphne, and other ASGI servers

The `socketio.ASGIApp` class is an ASGI compatible application that can forward Socket.IO traffic to an `socketio.AsyncServer` instance:

```python
sio = socketio.AsyncServer(async_mode='asgi')
app = socketio.ASGIApp(sio)
```

If desired, the `socketio.ASGIApp` class can forward any traffic that is not Socket.IO to another ASGI application, making it possible to deploy a standard ASGI web application and the Socket.IO server as a bundle:

```python
sio = socketio.AsyncServer(async_mode='asgi')
app = socketio.ASGIApp(sio, other_app)
```

The `ASGIApp` instance is a fully complaint ASGI instance that can be deployed with an ASGI compatible web server.

3.16.2 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:

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

3.16.3 Tornado

Tornado is a web framework with support for HTTP and WebSocket. Support for this framework requires Python 3.5 and newer. Only Tornado version 5 and newer are supported, thanks to its tight integration with asyncio.

Instances of class `socketio.AsyncServer` will automatically use tornado 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='tornado')
```

A server configured for tornado must include a request handler for Socket.IO:

```python
app = tornado.web.Application(
    [
        (r"/socket.io/", socketio.get_tornado_handler(sio)),
    ],
    # ... other application options
)
```

The tornado application can define other 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 tornado application is then executed in the usual manner:

```python
app.listen(port)
tornado.ioloop.IOLoop.current().start()
```

3.16.4 Sanic

Note: Due to some backward incompatible changes introduced in recent versions of Sanic, it is currently recommended that a Sanic application is deployed with the ASGI integration instead.

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 = Sanic()
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()
```

It has been reported that the CORS support provided by the Sanic extension sanic-cors is incompatible with this package's own support for this protocol. To disable CORS support in this package and let Sanic take full control, initialize the server as follows:

```
sio = socketio.AsyncServer(async_mode='sanic', cors_allowed_origins=[])
```

On the Sanic side you will need to enable the *CORS_SUPPORTS_CREDENTIALS* setting in addition to any other configuration that you use:

```
app.config['CORS_SUPPORTS_CREDENTIALS'] = True
```

3.16.5 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.WSGIApp`:

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

3.16.6 Eventlet with Gunicorn

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.

3.16.7 Gevent

Gevent is another asynchronous framework based on coroutines, very similar to eventlet. An Engine.IO server deployed with gevent has access to the long-polling and websocket transports.

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:

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

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

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

3.16.8 Gevent with Gunicorn

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
```

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.

3.16.9 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 asynchronous 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
```

3.16.10 Standard Threads

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.

Instances of class `socketio.Server` will automatically use the threading mode if neither eventlet nor gevent are 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.WSGIApp(sio, app.wsgi_app)

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

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

The example that follows shows how to start an Socket.IO application using Gunicorn's threaded worker class:

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

With the above configuration the server will be able to handle up to 100 concurrent clients.

When using standard threads, WebSocket is supported through the simple-websocket package, which must be installed separately. This package provides a multi-threaded WebSocket server that is compatible with Werkzeug and Gunicorn's threaded worker. Other multi-threaded web servers are not supported and will not enable the WebSocket transport.

3.17 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.

API REFERENCE

class socketio.**SimpleClient**(**args*, ***kwargs*)

A Socket.IO client.

This class implements a simple, yet fully compliant Socket.IO web client with support for websocket and long-polling transports.

The positional and keyword arguments given in the constructor are passed to the underlying *socketio.Client()* object.

call(*event*, *data=None*, *timeout=60*)

Emit an event to the server and wait for a response.

This method issues an emit and waits for the server to provide a response or acknowledgement. If the response does not arrive before the timeout, then a `TimeoutError` exception is raised.

> **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 server. Data can be of type `str`, `bytes`, `list` or `dict`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.
>
> - **timeout** – The waiting timeout. If the timeout is reached before the server acknowledges the event, then a `TimeoutError` exception is raised.

connect(*url*, *headers={}*, *auth=None*, *transports=None*, *namespace='/'*, *socketio_path='socket.io'*, *wait_timeout=5*)

Connect to a Socket.IO server.

> **Parameters**
>
> - **url** – The URL of the Socket.IO server. It can include custom query string parameters if required by the server. If a function is provided, the client will invoke it to obtain the URL each time a connection or reconnection is attempted.
>
> - **headers** – A dictionary with custom headers to send with the connection request. If a function is provided, the client will invoke it to obtain the headers dictionary each time a connection or reconnection is attempted.
>
> - **auth** – Authentication data passed to the server with the connection request, normally a dictionary with one or more string key/value pairs. If a function is provided, the client will invoke it to obtain the authentication data each time a connection or reconnection is attempted.

- **transports** – The list of allowed transports. Valid transports are `'polling'` and `'websocket'`. If not given, the polling transport is connected first, then an upgrade to websocket is attempted.

- **namespace** – The namespace to connect to as a string. If not given, the default namespace `/` is used.

- **socketio_path** – The endpoint where the Socket.IO server is installed. The default value is appropriate for most cases.

- **wait_timeout** – How long the client should wait for the connection to be established. The default is 5 seconds.

disconnect()

Disconnect from the server.

emit(*event*, *data=None*)

Emit an event to the server.

> **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 server. Data can be of type `str`, `bytes`, `list` or `dict`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.

This method schedules the event to be sent out and returns, without actually waiting for its delivery. In cases where the client needs to ensure that the event was received, *socketio.SimpleClient.call()* should be used instead.

receive(*timeout=None*)

Wait for an event from the server.

> **Parameters**
>
> **timeout** – The waiting timeout. If the timeout is reached before the server acknowledges the event, then a `TimeoutError` exception is raised.

The return value is a list with the event name as the first element. If the server included arguments with the event, they are returned as additional list elements.

property sid

The session ID received from the server.

The session ID is not guaranteed to remain constant throughout the life of the connection, as reconnections can cause it to change.

property transport

The name of the transport currently in use.

The transport is returned as a string and can be one of `polling` and `websocket`.

class socketio.**AsyncSimpleClient**(**args*, ***kwargs*)

A Socket.IO client.

This class implements a simple, yet fully compliant Socket.IO web client with support for websocket and long-polling transports.

The positional and keyword arguments given in the constructor are passed to the underlying *socketio.AsyncClient()* object.

async call(*event*, *data=None*, *timeout=60*)

Emit an event to the server and wait for a response.

This method issues an emit and waits for the server to provide a response or acknowledgement. If the response does not arrive before the timeout, then a TimeoutError exception is raised.

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 server. Data can be of type str, bytes, list or dict. To send multiple arguments, use a tuple where each element is of one of the types indicated above.

- **timeout** – The waiting timeout. If the timeout is reached before the server acknowledges the event, then a TimeoutError exception is raised.

Note: this method is a coroutine.

async connect(*url*, *headers={}*, *auth=None*, *transports=None*, *namespace='/'*, *socketio_path='socket.io'*, *wait_timeout=5*)

Connect to a Socket.IO server.

Parameters

- **url** – The URL of the Socket.IO server. It can include custom query string parameters if required by the server. If a function is provided, the client will invoke it to obtain the URL each time a connection or reconnection is attempted.

- **headers** – A dictionary with custom headers to send with the connection request. If a function is provided, the client will invoke it to obtain the headers dictionary each time a connection or reconnection is attempted.

- **auth** – Authentication data passed to the server with the connection request, normally a dictionary with one or more string key/value pairs. If a function is provided, the client will invoke it to obtain the authentication data each time a connection or reconnection is attempted.

- **transports** – The list of allowed transports. Valid transports are 'polling' and 'websocket'. If not given, the polling transport is connected first, then an upgrade to websocket is attempted.

- **namespace** – The namespace to connect to as a string. If not given, the default namespace / is used.

- **socketio_path** – The endpoint where the Socket.IO server is installed. The default value is appropriate for most cases.

- **wait_timeout** – How long the client should wait for the connection. The default is 5 seconds.

Note: this method is a coroutine.

async disconnect()

Disconnect from the server.

Note: this method is a coroutine.

async emit(*event*, *data=None*)

Emit an event to the server.

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 server. Data can be of type `str`, `bytes`, `list` or `dict`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.

Note: this method is a coroutine.

This method schedules the event to be sent out and returns, without actually waiting for its delivery. In cases where the client needs to ensure that the event was received, *socketio.SimpleClient.call()* should be used instead.

async receive(*timeout=None*)

Wait for an event from the server.

Parameters

timeout – The waiting timeout. If the timeout is reached before the server acknowledges the event, then a `TimeoutError` exception is raised.

Note: this method is a coroutine.

The return value is a list with the event name as the first element. If the server included arguments with the event, they are returned as additional list elements.

property sid

The session ID received from the server.

The session ID is not guaranteed to remain constant throughout the life of the connection, as reconnections can cause it to change.

property transport

The name of the transport currently in use.

The transport is returned as a string and can be one of `polling` and `websocket`.

class socketio.**Client**(*reconnection=True*, *reconnection_attempts=0*, *reconnection_delay=1*, *reconnection_delay_max=5*, *randomization_factor=0.5*, *logger=False*, *serializer='default'*, *json=None*, *handle_sigint=True*, ***kwargs*)

A Socket.IO client.

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

Parameters

- **reconnection** – `True` if the client should automatically attempt to reconnect to the server after an interruption, or `False` to not reconnect. The default is `True`.

- **reconnection_attempts** – How many reconnection attempts to issue before giving up, or 0 for infinite attempts. The default is 0.

- **reconnection_delay** – How long to wait in seconds before the first reconnection attempt. Each successive attempt doubles this delay.

- **reconnection_delay_max** – The maximum delay between reconnection attempts.

- **randomization_factor** – Randomization amount for each delay between reconnection attempts. The default is 0.5, which means that each delay is randomly adjusted by +/- 50%.

- **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 are logged even when `logger` is `False`.

- **serializer** – The serialization method to use when transmitting packets. Valid values are `'default'`, `'pickle'`, `'msgpack'` and `'cbor'`. Alternatively, a subclass of the `Packet` class with custom implementations of the `encode()` and `decode()` methods can be provided. Client and server must use compatible serializers.

- **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.

- **handle_sigint** – Set to `True` to automatically handle disconnection when the process is interrupted, or to `False` to leave interrupt handling to the calling application. Interrupt handling can only be enabled when the client instance is created in the main thread.

The Engine.IO configuration supports the following settings:

> **Parameters**
>
> - **request_timeout** – A timeout in seconds for requests. The default is 5 seconds.
>
> - **http_session** – an initialized `requests.Session` object to be used when sending requests to the server. Use it if you need to add special client options such as proxy servers, SSL certificates, etc.
>
> - **ssl_verify** – `True` to verify SSL certificates, or `False` to skip SSL certificate verification, allowing connections to servers with self signed certificates. 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`.

call(*event*, *data=None*, *namespace=None*, *timeout=60*)

Emit a custom event to the server and wait for the response.

This method issues an emit with a callback and waits for the callback to be invoked before returning. If the callback isn't invoked before the timeout, then a `TimeoutError` exception is raised. If the Socket.IO connection drops during the wait, this method still waits until the specified timeout.

> **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 server. Data can be of type `str`, `bytes`, `list` or `dict`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.
>
> - **namespace** – The Socket.IO namespace for the event. If this argument is omitted the event is emitted to the default namespace.
>
> - **timeout** – The waiting timeout. If the timeout is reached before the server acknowledges the event, then a `TimeoutError` exception is raised.

Note: this method is not thread safe. If multiple threads are emitting at the same time on the same client connection, messages composed of multiple packets may end up being sent in an incorrect sequence. Use standard concurrency solutions (such as a Lock object) to prevent this situation.

connect(*url*, *headers={}*, *auth=None*, *transports=None*, *namespaces=None*, *socketio_path='socket.io'*, *wait=True*, *wait_timeout=1*, *retry=False*)

Connect to a Socket.IO server.

> **Parameters**

- **url** – The URL of the Socket.IO server. It can include custom query string parameters if required by the server. If a function is provided, the client will invoke it to obtain the URL each time a connection or reconnection is attempted.

- **headers** – A dictionary with custom headers to send with the connection request. If a function is provided, the client will invoke it to obtain the headers dictionary each time a connection or reconnection is attempted.

- **auth** – Authentication data passed to the server with the connection request, normally a dictionary with one or more string key/value pairs. If a function is provided, the client will invoke it to obtain the authentication data each time a connection or reconnection is attempted.

- **transports** – The list of allowed transports. Valid transports are `'polling'` and `'websocket'`. If not given, the polling transport is connected first, then an upgrade to websocket is attempted.

- **namespaces** – The namespaces to connect as a string or list of strings. If not given, the namespaces that have registered event handlers are connected.

- **socketio_path** – The endpoint where the Socket.IO server is installed. The default value is appropriate for most cases.

- **wait** – if set to `True` (the default) the call only returns when all the namespaces are connected. If set to `False`, the call returns as soon as the Engine.IO transport is connected, and the namespaces will connect in the background.

- **wait_timeout** – How long the client should wait for the connection. The default is 1 second. This argument is only considered when `wait` is set to `True`.

- **retry** – Apply the reconnection logic if the initial connection attempt fails. The default is `False`.

Example usage:

```
sio = socketio.Client()
sio.connect('http://localhost:5000')
```

connected

    Indicates if the client is connected or not.

disconnect()

    Disconnect from the server.

emit(*event*, *data=None*, *namespace=None*, *callback=None*)

    Emit a custom event to the server.

        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 server. Data can be of type `str`, `bytes`, `list` or `dict`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.

- **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 server has received the message. The arguments that will be passed to the function are those provided by the server.

Note: this method is not thread safe. If multiple threads are emitting at the same time on the same client connection, messages composed of multiple packets may end up being sent in an incorrect sequence. Use standard concurrency solutions (such as a Lock object) to prevent this situation.

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:

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

The above example is equivalent to:

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

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

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

get_sid(*namespace=None*)

Return the `sid` associated with a connection.

> **Parameters**
>> **namespace** – The Socket.IO namespace. If this argument is omitted the handler is associated with the default namespace. Note that unlike previous versions, the current version of the Socket.IO protocol uses different `sid` values per namespace.

This method returns the `sid` for the requested namespace as a string.

namespaces

set of connected namespaces.

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. The `'*'` event name can be used to define a catch-all event handler.
> - **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. A catch-all namespace can be defined by passing `'*'` as the namespace.

Example usage:

```
# as a decorator:
@sio.on('connect')
def connect_handler():
    print('Connected!')

# as a method:
def message_handler(msg):
    print('Received message: ', msg)
    sio.send( 'response')
sio.on('message', message_handler)
```

The arguments passed to the handler function depend on the event type:

- The `'connect'` event handler does not take arguments.

- The `'disconnect'` event handler does not take arguments.

- The `'message'` handler and handlers for custom event names receive the message payload as only argument. Any values returned from a message handler will be passed to the client's acknowledgement callback function if it exists.

- A catch-all event handler receives the event name as first argument, followed by any arguments specific to the event.

- A catch-all namespace event handler receives the namespace as first argument, followed by any arguments specific to the event.

- A combined catch-all namespace and catch-all event handler receives the event name as first argument and the namespace as second argument, followed by any arguments specific to the event.

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.

send(*data*, *namespace=None*, *callback=None*)

Send a message to the server.

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

> **Parameters**
>
> > - **data** – The data to send to the server. Data can be of type `str`, `bytes`, `list` or `dict`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.
> >
> > - **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 server has received the message. The arguments that will be passed to the function are those provided by the server.

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.

transport()

Return the name of the transport used by the client.

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

wait()

Wait until the connection with the server ends.

Client applications can use this function to block the main thread during the life of the connection.

class socketio.**AsyncClient**(*reconnection=True*, *reconnection_attempts=0*, *reconnection_delay=1*, *reconnection_delay_max=5*, *randomization_factor=0.5*, *logger=False*, *serializer='default'*, *json=None*, *handle_sigint=True*, ***kwargs*)

A Socket.IO client for asyncio.

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

> **Parameters**
>
> - **reconnection** – `True` if the client should automatically attempt to reconnect to the server after an interruption, or `False` to not reconnect. The default is `True`.
>
> - **reconnection_attempts** – How many reconnection attempts to issue before giving up, or 0 for infinite attempts. The default is 0.
>
> - **reconnection_delay** – How long to wait in seconds before the first reconnection attempt. Each successive attempt doubles this delay.
>
> - **reconnection_delay_max** – The maximum delay between reconnection attempts.
>
> - **randomization_factor** – Randomization amount for each delay between reconnection attempts. The default is 0.5, which means that each delay is randomly adjusted by +/- 50%.
>
> - **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 are 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.
>
> - **handle_sigint** – Set to `True` to automatically handle disconnection when the process is interrupted, or to `False` to leave interrupt handling to the calling application. Interrupt handling can only be enabled when the client instance is created in the main thread.

The Engine.IO configuration supports the following settings:

Parameters

- **request_timeout** – A timeout in seconds for requests. The default is 5 seconds.

- **http_session** – an initialized `aiohttp.ClientSession` object to be used when sending requests to the server. Use it if you need to add special client options such as proxy servers, SSL certificates, etc.

- **ssl_verify** – `True` to verify SSL certificates, or `False` to skip SSL certificate verification, allowing connections to servers with self signed certificates. 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`.

async **call**(*event*, *data=None*, *namespace=None*, *timeout=60*)

Emit a custom event to the server and wait for the response.

This method issues an emit with a callback and waits for the callback to be invoked before returning. If the callback isn't invoked before the timeout, then a `TimeoutError` exception is raised. If the Socket.IO connection drops during the wait, this method still waits until the specified timeout.

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 server. Data can be of type `str`, `bytes`, `list` or `dict`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.

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

- **timeout** – The waiting timeout. If the timeout is reached before the server acknowledges the event, then a `TimeoutError` exception is raised.

Note: this method is not designed to be used concurrently. If multiple tasks are emitting at the same time on the same client connection, then messages composed of multiple packets may end up being sent in an incorrect sequence. Use standard concurrency solutions (such as a Lock object) to prevent this situation.

Note 2: this method is a coroutine.

async **connect**(*url*, *headers={}*, *auth=None*, *transports=None*, *namespaces=None*, *socketio_path='socket.io'*, *wait=True*, *wait_timeout=1*, *retry=False*)

Connect to a Socket.IO server.

Parameters

- **url** – The URL of the Socket.IO server. It can include custom query string parameters if required by the server. If a function is provided, the client will invoke it to obtain the URL each time a connection or reconnection is attempted.

- **headers** – A dictionary with custom headers to send with the connection request. If a function is provided, the client will invoke it to obtain the headers dictionary each time a connection or reconnection is attempted.

- **auth** – Authentication data passed to the server with the connection request, normally a dictionary with one or more string key/value pairs. If a function is provided, the client will invoke it to obtain the authentication data each time a connection or reconnection is attempted.

- **transports** – The list of allowed transports. Valid transports are `'polling'` and `'websocket'`. If not given, the polling transport is connected first, then an upgrade to websocket is attempted.

- **namespaces** – The namespaces to connect as a string or list of strings. If not given, the namespaces that have registered event handlers are connected.

- **socketio_path** – The endpoint where the Socket.IO server is installed. The default value is appropriate for most cases.

- **wait** – if set to `True` (the default) the call only returns when all the namespaces are connected. If set to `False`, the call returns as soon as the Engine.IO transport is connected, and the namespaces will connect in the background.

- **wait_timeout** – How long the client should wait for the connection. The default is 1 second. This argument is only considered when `wait` is set to `True`.

- **retry** – Apply the reconnection logic if the initial connection attempt fails. The default is `False`.

Note: this method is a coroutine.

Example usage:

```
sio = socketio.AsyncClient()
sio.connect('http://localhost:5000')
```

connected

Indicates if the client is connected or not.

async disconnect()

Disconnect from the server.

Note: this method is a coroutine.

async emit(*event*, *data=None*, *namespace=None*, *callback=None*)

Emit a custom event to the server.

> **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 server. Data can be of type `str`, `bytes`, `list` or `dict`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.
>
> - **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 server has received the message. The arguments that will be passed to the function are those provided by the server.

Note: this method is not designed to be used concurrently. If multiple tasks are emitting at the same time on the same client connection, then messages composed of multiple packets may end up being sent in an incorrect sequence. Use standard concurrency solutions (such as a Lock object) to prevent this situation.

Note 2: this method is a coroutine.

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:

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

>The above example is equivalent to:

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

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

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

get_sid(*namespace=None*)

>Return the sid associated with a connection.
>
>>**Parameters**
>>
>>>**namespace** – The Socket.IO namespace. If this argument is omitted the handler is associated with the default namespace. Note that unlike previous versions, the current version of the Socket.IO protocol uses different sid values per namespace.
>
>This method returns the sid for the requested namespace as a string.

namespaces

>set of connected namespaces.

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. The '*' event name can be used to define a catch-all event handler.
>>
>>- **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. A catch-all namespace can be defined by passing '*' as the namespace.
>
>Example usage:

```
# as a decorator:
@sio.on('connect')
def connect_handler():
    print('Connected!')
```

```python
# as a method:
def message_handler(msg):
    print('Received message: ', msg)
    sio.send( 'response')
sio.on('message', message_handler)
```

The arguments passed to the handler function depend on the event type:

- The `'connect'` event handler does not take arguments.

- The `'disconnect'` event handler does not take arguments.

- The `'message'` handler and handlers for custom event names receive the message payload as only argument. Any values returned from a message handler will be passed to the client's acknowledgement callback function if it exists.

- A catch-all event handler receives the event name as first argument, followed by any arguments specific to the event.

- A catch-all namespace event handler receives the namespace as first argument, followed by any arguments specific to the event.

- A combined catch-all namespace and catch-all event handler receives the event name as first argument and the namespace as second argument, followed by any arguments specific to the event.

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.

async send(*data*, *namespace=None*, *callback=None*)

Send a message to the server.

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

> **Parameters**
>
> - **data** – The data to send to the server. Data can be of type `str`, `bytes`, `list` or `dict`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.
>
> - **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 server has received the message. The arguments that will be passed to the function are those provided by the server.

Note: this method is a coroutine.

async 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.
> - `args` – arguments to pass to the function.
> - `kwargs` – keyword arguments to pass to the function.

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

`transport`()

Return the name of the transport used by the client.

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

`async wait`()

Wait until the connection with the server ends.

Client applications can use this function to block the main thread during the life of the connection.

Note: this method is a coroutine.

class `socketio.Server`(*client_manager=None*, *logger=False*, *serializer='default'*, *json=None*, *async_handlers=True*, *always_connect=False*, *namespaces=None*, ***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`. Note that fatal errors are logged even when `logger` is `False`.
> - `serializer` – The serialization method to use when transmitting packets. Valid values are `'default'`, `'pickle'`, `'msgpack'` and `'cbor'`. Alternatively, a subclass of the `Packet` class with custom implementations of the `encode()` and `decode()` methods can be provided. Client and server must use compatible serializers.
> - `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 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`.

- **namespaces** – a list of namespaces that are accepted, in addition to any namespaces for which handlers have been defined. The default is *['/']*, which always accepts connections to the default namespace. Set to '*' to accept all namespaces.

- **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_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 20 seconds.

- **max_http_buffer_size** – The maximum size that is accepted for incoming messages. The default is 1,000,000 bytes. In spite of its name, the value set in this argument is enforced for HTTP long-polling and WebSocket connections.

- **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 tot he 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`.

call(*event*, *data=None*, *to=None*, *sid=None*, *namespace=None*, *timeout=60*, *ignore_queue=False*)

Emit a custom event to a client and wait for the response.

This method issues an emit with a callback and waits for the callback to be invoked before returning. If the callback isn't invoked before the timeout, then a `TimeoutError` exception is raised. If the Socket.IO connection drops during the wait, this method still waits until the specified timeout.

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`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.

- **to** – The session ID of the recipient client.

- **sid** – Alias for the `to` parameter.

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

- **timeout** – The waiting timeout. If the timeout is reached before the client acknowledges the event, then a `TimeoutError` exception is raised.

- **ignore_queue** – Only used when a message queue is configured. If set to `True`, the event is emitted to the client 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 not thread safe. If multiple threads are emitting at the same time to the same client, then messages composed of multiple packets may end up being sent in an incorrect sequence. Use standard concurrency solutions (such as a Lock object) to prevent this situation.

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*, *ignore_queue=False*)

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.

- **ignore_queue** – Only used when a message queue is configured. If set to `True`, the disconnect is processed locally, without broadcasting on the queue. It is recommended to always leave this parameter with its default value of `False`.

emit(*event*, *data=None*, *to=None*, *room=None*, *skip_sid=None*, *namespace=None*, *callback=None*, *ignore_queue=False*)

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`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.

- **to** – The recipient of the message. This can be set to the session ID of a client to address only that client, to any custom room created by the application to address all the clients in that room, or to a list of custom room names. If this argument is omitted the event is broadcasted to all connected clients.

- **room** – Alias for the `to` parameter.

- **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. To skip multiple sids, pass a list.

- **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 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 not thread safe. If multiple threads are emitting at the same time to the same client, then messages composed of multiple packets may end up being sent in an incorrect sequence. Use standard concurrency solutions (such as a Lock object) to prevent this situation.

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.

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:

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

The above example is equivalent to:

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

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

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

get_environ(*sid*, *namespace=None*)

    Return the WSGI environ dictionary for a client.

        Parameters

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

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

get_session(*sid*, *namespace=None*)

    Return the user session for a client.

        Parameters

- **sid** – The session id of the client.

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

    The return value is a dictionary. Modifications made to this dictionary are not guaranteed to be preserved unless save_session() is called, or when the session context manager 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.

instrument(*auth=None*, *mode='development'*, *read_only=False*, *server_id=None*, *namespace='/admin'*, *server_stats_interval=2*)

    Instrument the Socket.IO server for monitoring with the Socket.IO Admin UI.

        Parameters

- **auth** – Authentication credentials for Admin UI access. Set to a dictionary with the expected login (usually username and password) or a list of dictionaries if more than one set of credentials need to be available. For more complex authentication methods, set to a callable that receives the authentication dictionary as an argument and returns True if the user is allowed or False otherwise. To disable authentication, set this argument to False (not recommended, never do this on a production server).

- **mode** – The reporting mode. The default is 'development', which is best used while debugging, as it may have a significant performance effect. Set to 'production' to reduce the amount of information that is reported to the admin UI.

- **read_only** – If set to True, the admin interface will be read-only, with no option to modify room assignments or disconnect clients. The default is False.

- **server_id** – The server name to use for this server. If this argument is omitted, the server generates its own name.

- **namespace** – The Socket.IO namespace to use for the admin interface. The default is /admin.

- **server_stats_interval** – The interval in seconds at which the server emits a summary of it stats to all connected admins.

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. The '*' event name can be used to define a catch-all event handler.
>
> - **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. A catch-all namespace can be defined by passing '*' as the namespace.

Example usage:

```python
# as a decorator:
@sio.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)
    sio.send(sid, 'response')
socket_io.on('message', namespace='/chat', handler=message_handler)
```

The arguments passed to the handler function depend on the event type:

- The 'connect' event handler receives the sid (session ID) for the client and the WSGI environment dictionary as arguments.

- The 'disconnect' handler receives the sid for the client as only argument.

- The `'message'` handler and handlers for custom event names receive the `sid` for the client and the message payload as arguments. Any values returned from a message handler will be passed to the client's acknowledgement callback function if it exists.

- A catch-all event handler receives the event name as first argument, followed by any arguments specific to the event.

- A catch-all namespace event handler receives the namespace as first argument, followed by any arguments specific to the event.

- A combined catch-all namespace and catch-all event handler receives the event name as first argument and the namespace as second argument, followed by any arguments specific to the event.

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.

save_session(*sid*, *session*, *namespace=None*)

    Store the user session for a client.

        Parameters

            - **sid** – The session id of the client.

            - **session** – The session dictionary.

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

send(*data*, *to=None*, *room=None*, *skip_sid=None*, *namespace=None*, *callback=None*, *ignore_queue=False*)

    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`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.

            - **to** – The recipient of the message. This can be set to the session ID of a client to address only that client, to any any custom room created by the application to address all the clients in that room, or to a list of custom room names. If this argument is omitted the event is broadcasted to all connected clients.

            - **room** – Alias for the `to` parameter.

            - **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. To skip multiple sids, pass a list.

- **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 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`.

session(*sid*, *namespace=None*)

Return the user session for a client with context manager syntax.

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

This is a context manager that returns the user session dictionary for the client. Any changes that are made to this dictionary inside the context manager block are saved back to the session. Example usage:

```python
@sio.on('connect')
def on_connect(sid, environ):
    username = authenticate_user(environ)
    if not username:
        return False
    with sio.session(sid) as session:
        session['username'] = username

@sio.on('message')
def on_message(sid, msg):
    with sio.session(sid) as session:
        print('received message from ', session['username'])
```

shutdown()

Stop Socket.IO background tasks.

This method stops all background activity initiated by the Socket.IO server. It must be called before shutting down the web server.

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.

transport(*sid*, *namespace=None*)

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

class socketio.**AsyncServer**(*client_manager=None*, *logger=False*, *json=None*, *async_handlers=True*, *namespaces=None*, ***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.

> **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`. Note that fatal errors are 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.
>
> - **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`.
>
> - **namespaces** – a list of namespaces that are accepted, in addition to any namespaces for which handlers have been defined. The default is *['/']*, which always accepts connections to the default namespace. Set to '*' to accept all namespaces.
>
> - **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", "sanic", "tornado" and "asgi". If this argument is not given, "aiohttp" is tried first, followed by "sanic", "tornado", and finally "asgi". The first async mode that has all its dependencies installed is the 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 20 seconds.

- **max_http_buffer_size** – The maximum size that is accepted for incoming messages. The default is 1,000,000 bytes. In spite of its name, the value set in this argument is enforced for HTTP long-polling and WebSocket connections.

- **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`.

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

Attach the Socket.IO server to an application.

async call(*event*, *data=None*, *to=None*, *sid=None*, *namespace=None*, *timeout=60*, *ignore_queue=False*)

Emit a custom event to a client and wait for the response.

This method issues an emit with a callback and waits for the callback to be invoked before returning. If the callback isn't invoked before the timeout, then a `TimeoutError` exception is raised. If the Socket.IO connection drops during the wait, this method still waits until the specified timeout.

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`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.

- **to** – The session ID of the recipient client.

- **sid** – Alias for the `to` parameter.

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

- **timeout** – The waiting timeout. If the timeout is reached before the client acknowledges the event, then a `TimeoutError` exception is raised.

- **ignore_queue** – Only used when a message queue is configured. If set to `True`, the event is emitted to the client 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 not designed to be used concurrently. If multiple tasks are emitting at the same time to the same client connection, then messages composed of multiple packets may end up being sent in an incorrect sequence. Use standard concurrency solutions (such as a Lock object) to prevent this situation.

Note 2: this method is a coroutine.

async **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.

async **disconnect**(*sid*, *namespace=None*, *ignore_queue=False*)

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.
>
> - **ignore_queue** – Only used when a message queue is configured. If set to `True`, the disconnect is processed locally, without broadcasting on the queue. It is recommended to always leave this parameter with its default value of `False`.

Note: this method is a coroutine.

async **emit**(*event*, *data=None*, *to=None*, *room=None*, *skip_sid=None*, *namespace=None*, *callback=None*, *ignore_queue=False*)

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`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.
>
> - **to** – The recipient of the message. This can be set to the session ID of a client to address only that client, to any any custom room created by the application to address all the clients in that room, or to a list of custom room names. If this argument is omitted the event is broadcasted to all connected clients.

- **room** – Alias for the `to` parameter.

- **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 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 not designed to be used concurrently. If multiple tasks are emitting at the same time to the same client connection, then messages composed of multiple packets may end up being sent in an incorrect sequence. Use standard concurrency solutions (such as a Lock object) to prevent this situation.

Note 2: this method is a coroutine.

async 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.

Note: this method is a coroutine.

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:

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

The above example is equivalent to:

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

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

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

get_environ(*sid*, *namespace=None*)

    Return the WSGI environ dictionary for a client.

        Parameters

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

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

async get_session(*sid*, *namespace=None*)

    Return the user session for a client.

        Parameters

- **sid** – The session id of the client.

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

    The return value is a dictionary. Modifications made to this dictionary are not guaranteed to be preserved. If you want to modify the user session, use the `session` context manager instead.

async 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.

instrument(*auth=None*, *mode='development'*, *read_only=False*, *server_id=None*, *namespace='/admin'*, *server_stats_interval=2*)

    Instrument the Socket.IO server for monitoring with the Socket.IO Admin UI.

        Parameters

- **auth** – Authentication credentials for Admin UI access. Set to a dictionary with the expected login (usually `username` and `password`) or a list of dictionaries if more than one set of credentials need to be available. For more complex authentication methods, set to a callable that receives the authentication dictionary as an argument and returns `True` if the user is allowed or `False` otherwise. To disable authentication, set this argument to `False` (not recommended, never do this on a production server).

- **mode** – The reporting mode. The default is `'development'`, which is best used while debugging, as it may have a significant performance effect. Set to `'production'` to reduce the amount of information that is reported to the admin UI.

- **read_only** – If set to `True`, the admin interface will be read-only, with no option to modify room assignments or disconnect clients. The default is `False`.

- **server_id** – The server name to use for this server. If this argument is omitted, the server generates its own name.

- **namespace** – The Socket.IO namespace to use for the admin interface. The default is `/admin`.

- **server_stats_interval** – The interval in seconds at which the server emits a summary of it stats to all connected admins.

async 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.

Note: this method is a coroutine.

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. The `'*'` event name can be used to define a catch-all event handler.
>
> - **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. A catch-all namespace can be defined by passing `'*'` as the namespace.

Example usage:

```python
# as a decorator:
@sio.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)
    sio.send(sid, 'response')
socket_io.on('message', namespace='/chat', handler=message_handler)
```

The arguments passed to the handler function depend on the event type:

- The `'connect'` event handler receives the `sid` (session ID) for the client and the WSGI environment dictionary as arguments.

- The `'disconnect'` handler receives the `sid` for the client as only argument.

- The `'message'` handler and handlers for custom event names receive the `sid` for the client and the message payload as arguments. Any values returned from a message handler will be passed to the client's acknowledgement callback function if it exists.

- A catch-all event handler receives the event name as first argument, followed by any arguments specific to the event.

- A catch-all namespace event handler receives the namespace as first argument, followed by any arguments specific to the event.

- A combined catch-all namespace and catch-all event handler receives the event name as first argument and the namespace as second argument, followed by any arguments specific to the event.

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.

async save_session(*sid*, *session*, *namespace=None*)

Store the user session for a client.

>**Parameters**
>
>- **sid** – The session id of the client.
>
>- **session** – The session dictionary.
>
>- **namespace** – The Socket.IO namespace. If this argument is omitted the default namespace is used.

async send(*data*, *to=None*, *room=None*, *skip_sid=None*, *namespace=None*, *callback=None*, *ignore_queue=False*)

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`. To send multiple arguments, use a tuple where each element is of one of the types indicated above.
>
>- **to** – The recipient of the message. This can be set to the session ID of a client to address only that client, to any any custom room created by the application to address all the clients in that room, or to a list of custom room names. If this argument is omitted the event is broadcasted to all connected clients.
>
>- **room** – Alias for the `to` parameter.
>
>- **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 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.

session(*sid*, *namespace=None*)

Return the user session for a client with context manager syntax.

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

This is a context manager that returns the user session dictionary for the client. Any changes that are made to this dictionary inside the context manager block are saved back to the session. Example usage:

```python
@eio.on('connect')
def on_connect(sid, environ):
    username = authenticate_user(environ)
    if not username:
        return False
    with eio.session(sid) as session:
        session['username'] = username

@eio.on('message')
def on_message(sid, msg):
    async with eio.session(sid) as session:
        print('received message from ', session['username'])
```

async shutdown()

Stop Socket.IO background tasks.

This method stops all background activity initiated by the Socket.IO server. It must be called before shutting down the web server.

async 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.

> **transport**(*sid*, *namespace=None*)
>
>> 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.
>>>
>>> - **namespace** – The Socket.IO namespace. If this argument is omitted the default namespace is used.

class socketio.exceptions.**ConnectionRefusedError**(**args*)

> Connection refused exception.
>
> This exception can be raised from a connect handler when the connection is not accepted. The positional arguments provided with the exception are returned with the error packet to the client.

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

> WSGI middleware for Socket.IO.
>
> This middleware dispatches traffic to a Socket.IO application. It can also serve a list of static files to the client, or forward unrelated HTTP traffic to another WSGI application.
>
>> **Parameters**
>>
>> - **socketio_app** – The Socket.IO server. Must be an instance of the `socketio.Server` class.
>>
>> - **wsgi_app** – The WSGI app that receives all other traffic.
>>
>> - **static_files** – A dictionary with static file mapping rules. See the documentation for details on this argument.
>>
>> - **socketio_path** – The endpoint where the Socket.IO application should be installed. The default value is appropriate for most cases.
>
> Example usage:

```python
import socketio
import eventlet
from . import wsgi_app

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

class socketio.**ASGIApp**(*socketio_server*, *other_asgi_app=None*, *static_files=None*, *socketio_path='socket.io'*, *on_startup=None*, *on_shutdown=None*)

> ASGI application middleware for Socket.IO.
>
> This middleware dispatches traffic to an Socket.IO application. It can also serve a list of static files to the client, or forward unrelated HTTP traffic to another ASGI application.
>
>> **Parameters**
>>
>> - **socketio_server** – The Socket.IO server. Must be an instance of the `socketio.AsyncServer` class.
>>
>> - **static_files** – A dictionary with static file mapping rules. See the documentation for details on this argument.
>>
>> - **other_asgi_app** – A separate ASGI 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. With a value of `None`, all incoming traffic is directed to the Socket.IO server, with the assumption that routing, if necessary, is handled by a different layer. When this option is set to `None`, `static_files` and `other_asgi_app` are ignored.

- **on_startup** – function to be called on application startup; can be coroutine

- **on_shutdown** – function to be called on application shutdown; can be coroutine

Example usage:

```python
import socketio
import uvicorn

sio = socketio.AsyncServer()
app = socketio.ASGIApp(sio, static_files={
    '/': 'index.html',
    '/static': './public',
})
uvicorn.run(app, host='127.0.0.1', port=5000)
```

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

This class has been renamed to WSGIApp and is now deprecated.

class socketio.**ClientNamespace**(*namespace=None*)

Base class for client-side 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.

call(*event*, *data=None*, *namespace=None*, *timeout=None*)

Emit a custom event to the server and wait for the response.

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

disconnect()

Disconnect from the server.

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

emit(*event*, *data=None*, *namespace=None*, *callback=None*)

Emit a custom event to the server.

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

send(*data*, *room=None*, *namespace=None*, *callback=None*)

Send a message to the server.

The only difference with the *socketio.Client.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 overridden if special dispatching rules are needed, or if having a single method that catches all events is desired.

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

> Base class for server-side 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.

> **call**(*event*, *data=None*, *to=None*, *sid=None*, *namespace=None*, *timeout=None*, *ignore_queue=False*)
>
> > Emit a custom event to a client and wait for the response.
> >
> > The only difference with the *socketio.Server.call()* method is that when the namespace argument is not given the namespace associated with the class 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*, *to=None*, *room=None*, *skip_sid=None*, *namespace=None*, *callback=None*, *ignore_queue=False*)
>
> > 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.

> **get_session**(*sid*, *namespace=None*)
>
> > Return the user session for a client.
> >
> > The only difference with the *socketio.Server.get_session()* 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.

save_session(*sid*, *session*, *namespace=None*)

> Store the user session for a client.
>
> The only difference with the `socketio.Server.save_session()` method is that when the `namespace` argument is not given the namespace associated with the class is used.

send(*data*, *to=None*, *room=None*, *skip_sid=None*, *namespace=None*, *callback=None*, *ignore_queue=False*)

> 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.

session(*sid*, *namespace=None*)

> Return the user session for a client with context manager syntax.
>
> The only difference with the `socketio.Server.session()` 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 overridden if special dispatching rules are needed, or if having a single method that catches all events is desired.

class socketio.**AsyncClientNamespace**(*namespace=None*)

Base class for asyncio client-side 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.

async call(*event*, *data=None*, *namespace=None*, *timeout=None*)

> Emit a custom event to the server and wait for the response.
>
> The only difference with the `socketio.Client.call()` method is that when the `namespace` argument is not given the namespace associated with the class is used.

async disconnect()

> Disconnect a client.
>
> The only difference with the `socketio.Client.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.

async emit(*event*, *data=None*, *namespace=None*, *callback=None*)

> Emit a custom event to the server.
>
> The only difference with the `socketio.Client.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.

async send(*data*, *namespace=None*, *callback=None*)

Send a message to the server.

The only difference with the `socketio.Client.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.

async 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.

Note: this method is a coroutine.

class socketio.**AsyncNamespace**(*namespace=None*)

Base class for asyncio server-side 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.

async call(*event*, *data=None*, *to=None*, *sid=None*, *namespace=None*, *timeout=None*, *ignore_queue=False*)

Emit a custom event to a client and wait for the response.

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

async 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.

async 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.

async emit(*event*, *data=None*, *to=None*, *room=None*, *skip_sid=None*, *namespace=None*, *callback=None*, *ignore_queue=False*)

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.

async 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.

Note: this method is a coroutine.

async get_session(*sid*, *namespace=None*)

Return the user session for a client.

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

Note: this method is a coroutine.

async 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.

Note: this method is a coroutine.

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.

async save_session(*sid*, *session*, *namespace=None*)

Store the user session for a client.

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

Note: this method is a coroutine.

async send(*data*, *to=None*, *room=None*, *skip_sid=None*, *namespace=None*, *callback=None*, *ignore_queue=False*)

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.

session(*sid*, *namespace=None*)

Return the user session for a client with context manager syntax.

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

async 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.

Note: this method is a coroutine.

class socketio.**Manager**

>   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**(*eio_sid*, *namespace*)

>   >   Register a client connection to a namespace.

>   **disconnect**(*sid*, *namespace*, ***kwargs*)

>   >   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*, *eio_sid=None*)

>   >   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*, *id*, *data*)

>   >   Invoke an application callback.

class socketio.**PubSubManager**(*channel='socketio'*, *write_only=False*, *logger=None*)

>   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.

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

>   >   Remove all participants from a room.

---

connect(*eio_sid*, *namespace*)

> Register a client connection to a namespace.

disconnect(*sid*, *namespace=None*, ***kwargs*)

> Register a client disconnect from a namespace.

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()`.

enter_room(*sid*, *namespace*, *room*, *eio_sid=None*)

> 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*, *id*, *data*)

> Invoke an application callback.

class socketio.**KombuManager**(*url='amqp://guest:guest@localhost:5672//'*, *channel='socketio'*, *write_only=False*, *logger=None*, *connection_options=None*, *exchange_options=None*, *queue_options=None*, *producer_options=None*)

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 to `True`, only initialize to emit events. The default of `False` initializes the class for emitting and receiving.
- **connection_options** – additional keyword arguments to be passed to `kombu.Connection()`.
- **exchange_options** – additional keyword arguments to be passed to `kombu.Exchange()`.
- **queue_options** – additional keyword arguments to be passed to `kombu.Queue()`.
- **producer_options** – additional keyword arguments to be passed to `kombu.Producer()`.

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

Remove all participants from a room.

connect(*eio_sid*, *namespace*)

Register a client connection to a namespace.

disconnect(*sid*, *namespace=None*, ***kwargs*)

Register a client disconnect from a namespace.

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()`.

enter_room(*sid*, *namespace*, *room*, *eio_sid=None*)

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*, *id*, *data*)

Invoke an application callback.

class socketio.**RedisManager**(*url='redis://localhost:6379/0'*, *channel='socketio'*, *write_only=False*, *logger=None*, *redis_options=None*)

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://`. To use an SSL connection, use `rediss://`.
>
> - **channel** – The channel name on which the server sends and receives notifications. Must be the same in all the servers.
>
> - **write_only** – If set to `True`, only initialize to emit events. The default of `False` initializes the class for emitting and receiving.
>
> - **redis_options** – additional keyword arguments to be passed to `Redis.from_url()`.

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

Remove all participants from a room.

connect(*eio_sid*, *namespace*)

Register a client connection to a namespace.

disconnect(*sid*, *namespace=None*, ***kwargs*)

Register a client disconnect from a namespace.

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()*.

enter_room(*sid*, *namespace*, *room*, *eio_sid=None*)

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*, *id*, *data*)

> Invoke an application callback.

class socketio.**KafkaManager**(*url='kafka://localhost:9092'*, *channel='socketio'*, *write_only=False*)

> Kafka based client manager.
>
> This class implements a Kafka backend for event sharing across multiple processes.
>
> To use a Kafka backend, initialize the *Server* instance as follows:

```
url = 'kafka://hostname:port'
server = socketio.Server(client_manager=socketio.KafkaManager(url))
```

> **Parameters**
>
> - **url** – The connection URL for the Kafka server. For a default Kafka store running on the same host, use `kafka://`. For a highly available deployment of Kafka, pass a list with all the connection URLs available in your cluster.
> - **channel** – The channel name (topic) on which the server sends and receives notifications. Must be the same in all the servers.
> - **write_only** – If set to `True`, only initialize to emit events. The default of `False` initializes the class for emitting and receiving.

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

> Remove all participants from a room.

connect(*eio_sid*, *namespace*)

> Register a client connection to a namespace.

disconnect(*sid*, *namespace=None*, ***kwargs*)

> Register a client disconnect from a namespace.

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()*.

enter_room(*sid*, *namespace*, *room*, *eio_sid=None*)

> 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*, *id*, *data*)

Invoke an application callback.

class socketio.**AsyncManager**

Manage a client list for an asyncio server.

async close_room(*room*, *namespace*)

Remove all participants from a room.

Note: this method is a coroutine.

async connect(*eio_sid*, *namespace*)

Register a client connection to a namespace.

Note: this method is a coroutine.

async disconnect(*sid*, *namespace*, ***kwargs*)

Disconnect a client.

Note: this method is a coroutine.

async 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.

async enter_room(*sid*, *namespace*, *room*, *eio_sid=None*)

Add a client to a room.

Note: this method is a coroutine.

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.

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

>   Remove a client from a room.

>   Note: this method is a coroutine.

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.

async **trigger_callback**(*sid*, *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*, *logger=None*, *redis_options=None*)

>   Redis based client manager for asyncio servers.

>   This class implements a Redis backend for event sharing across multiple processes.

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

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

>   **Parameters**

>   - **url** – The connection URL for the Redis server. For a default Redis store running on the same host, use `redis://`. To use an SSL connection, use `rediss://`.

>   - **channel** – The channel name on which the server sends and receives notifications. Must be the same in all the servers.

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

>   - **redis_options** – additional keyword arguments to be passed to `aioredis.from_url()`.

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

>   Remove all participants from a room.

>   Note: this method is a coroutine.

async **connect**(*eio_sid*, *namespace*)

>   Register a client connection to a namespace.

>   Note: this method is a coroutine.

async **disconnect**(*sid*, *namespace*, ***kwargs*)

>   Disconnect a client.

>   Note: this method is a coroutine.

async **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()`.

Note: this method is a coroutine.

async enter_room(*sid*, *namespace*, *room*, *eio_sid=None*)

Add a client to a room.

Note: this method is a coroutine.

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.

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

Remove a client from a room.

Note: this method is a coroutine.

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.

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

Invoke an application callback.

Note: this method is a coroutine.

class socketio.**AsyncAioPikaManager**(*url='amqp://guest:guest@localhost:5672//'*, *channel='socketio'*, *write_only=False*, *logger=None*)

Client manager that uses aio_pika for inter-process messaging under asyncio.

This class implements a client manager backend for event sharing across multiple processes, using RabbitMQ

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

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

Parameters

- **url** – The connection URL for the backend messaging queue. Example connection URLs are `'amqp://guest:guest@localhost:5672//'` for RabbitMQ.

- **channel** – The channel name on which the server sends and receives notifications. Must be the same in all the servers. With this manager, the channel name is the exchange name in rabbitmq

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

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

    Remove all participants from a room.

    Note: this method is a coroutine.

async connect(*eio_sid*, *namespace*)

    Register a client connection to a namespace.

    Note: this method is a coroutine.

async disconnect(*sid*, *namespace*, ***kwargs*)

    Disconnect a client.

    Note: this method is a coroutine.

async 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()`.

    Note: this method is a coroutine.

async enter_room(*sid*, *namespace*, *room*, *eio_sid=None*)

    Add a client to a room.

    Note: this method is a coroutine.

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.

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

    Remove a client from a room.

    Note: this method is a coroutine.

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.

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

    Invoke an application callback.

    Note: this method is a coroutine.

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

socketio, 37

W