python-engineio Documentation

Miguel Grinberg

Oct 16, 2024

CONTENTS

This project implements Python based Engine.IO client and server that can run standalone or integrated with a variety of Python web frameworks and applications.

CHAPTER

ONE

GETTING STARTED

1.1 What is Engine.IO?

Engine.IO is a lightweight 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.

The Engine.IO protocol is extremely simple. Once a connection between a client and a server is established, either side can send "messages" to the other side. Event handlers provided by the applications on both ends are invoked when a message is received, or when a connection is established or dropped.

1.2 Client Examples

The example that follows shows a simple Python client:

```
import engineio

eio = engineio.Client()

@eio.on('connect')
def on_connect():
    print('connection established')

@eio.on('message')
def on_message(data):
    print('message received with ', data)
    eio.send({'response': 'my response'})

@eio.on('disconnect')
def on_disconnect():
    print('disconnected from server')

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

And here is a similar client written using the official Engine.IO Javascript client:

```
<script src="/path/to/engine.io.js"></script>
<script>
```

(continues on next page)

(continued from previous page)

```
    var socket = eio('http://localhost:5000');
    socket.on('open', function() { console.log('connection established'); });
    socket.on('message', function(data) {
        console.log('message received with ' + data);
        socket.send({response: 'my response'});
    });
    socket.on('close', function() { console.log('disconnected from server'); });
</script>
```

1.3 Client Features

- Can connect to other Engine.IO complaint servers besides the one in this package.
- 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.

1.4 Server Examples

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

```
import engineio
import eventlet

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

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

@eio.on('message')
def message(sid, data):
    print("message ", data)
    eio.send(sid, 'reply')

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

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

Below is a similar application, coded for asyncio and the Uvicorn web server:

```
import engineio
import uvicorn

eio = engineio.AsyncServer()
app = engineio.ASGIApp(eio, static_files={
    '/': {'content_type': 'text/html', 'filename': 'index.html'}
})

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

@eio.on('message')
async def message(sid, data):
    print("message ", data)
    await eio.send(sid, 'reply')

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

if __name__ == '__main__':
    uvicorn.run('127.0.0.1', 5000)
```

1.5 Server Features

- Can accept clients running other complaint Engine.IO clients besides the one in this package.
- 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 and ASGI web servers includind 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.
- Uses an event-based architecture implemented with decorators that hides the details of the protocol.
- Implements HTTP long-polling and WebSocket transports.
- Supports XHR2 and XHR browsers as clients.
- Supports text and binary messages.
- Supports gzip and deflate HTTP compression.
- Configurable CORS responses to avoid cross-origin problems with browsers.

CHAPTER

TWO

THE ENGINE.IO CLIENT

This package contains two Engine.IO clients:

- The *engineio.Client()* class creates a client compatible with the standard Python library.
- The *engineio.AsyncClient()* class creates a client compatible with the `asyncio` package.

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

2.1 Installation

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

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

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

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

2.2 Creating a Client Instance

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

```
import engineio

# standard Python
eio = engineio.Client()

# asyncio
eio = engineio.AsyncClient()
```

2.3 Defining Event Handlers

To responds to events triggered by the connection or the server, event Handler functions must be defined using the `on` decorator:

```
@eio.on('connect')
def on_connect():
    print('I'm connected!')

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

@eio.on('disconnect')
def on_disconnect():
    print('I'm disconnected!')
```

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

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

The argument given to the `on` decorator is the event name. The events that are supported are `connect`, `message` and `disconnect`. Note that the `disconnect` handler is invoked for application initiated disconnects, server initiated disconnects, or accidental disconnects, for example due to networking failures.

The `data` argument passed to the `'message'` event handler contains application-specific data provided by the server with the event.

2.4 Connecting to a Server

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

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

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

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

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

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

2.5 Sending Messages

The client can send a message to the server using the `send()` method:

```
eio.send({'foo': 'bar'})
```

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

```
await eio.send({'foo': 'bar'})
```

The single argument provided to the method is the data that is passed on to the server. The data can be of type `str`, `bytes`, `dict` or `list`. The data included inside dictionaries and lists is also constrained to these types.

The `send()` 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.

2.6 Disconnecting from the Server

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

```
eio.disconnect()
```

For the `asyncio` client this is a coroutine:

```
await eio.disconnect()
```

2.7 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 Engine.IO client.

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

```
eio.wait()
```

Or in the `asyncio` version:

```
await eio.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

eio.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:

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

eio.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, but the background function is.

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

```
eio.sleep(2)
```

Or for `asyncio`:

```
await eio.sleep(2)
```

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

2.8 Debugging and Troubleshooting

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

```
import engineio

# standard Python
eio = engineio.Client(logger=True)

# asyncio
eio = engineio.AsyncClient(logger=True)
```

The `logger` argument 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.

CHAPTER

THREE

THE ENGINE.IO SERVER

This package contains two Engine.IO servers:

- The *engineio.Server()* class creates a server compatible with the standard Python library.
- The *engineio.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 Python Engine.IO server use the following command:

```
pip install "python-engineio"
```

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

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

```
import engineio

# create a Engine.IO server
eio = engineio.Server()

# wrap with a WSGI application
app = engineio.WSGIApp(eio)
```

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

```
# create a Engine.IO server
eio = engineio.AsyncServer()

# wrap with ASGI application
app = engineio.ASGIApp(eio)
```

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

```
from wsgi import app  # a Flask, Django, etc. application
app = engineio.WSGIApp(eio, app)
```

3.3 Serving Static Files

The Engine.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:

```
static_files = {
    '/': 'latency.html',
    '/static/engine.io.js': 'static/engine.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:

```
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:

```
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:

```
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 `engineio.WSGIApp` or `engineio.ASGIApp` classes:

```
# for standard WSGI applications
eio = engineio.Server()
app = engineio.WSGIApp(eio, static_files=static_files)

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

The routing precedence in these two classes is as follows:

- First, the path is checked against the Engine.IO path.
- Next, the path is checked against the static file configuration, if present.
- If the path did not match the Engine.IO path 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

To responds to events triggered by the connection or the client, event Handler functions must be defined using the `on` decorator:

```
@eio.on('connect')
def on_connect(sid):
    print('A client connected!')

@eio.on('message')
def on_message(sid, data):
    print('I received a message!')

@eio.on('disconnect')
def on_disconnect(sid):
    print('Client disconnected!')
```

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

```
@eio.on('message')
async def on_message(sid, data):
    print('I received a message!')
```

The argument given to the `on` decorator is the event name. The events that are supported are `connect`, `message` and `disconnect`. Note that the `disconnect` handler is invoked for client initiated disconnects, server initiated disconnects, or accidental disconnects, for example due to networking failures.

The `sid` argument passed into all the event handlers is a connection identifier for the client. All the events from a client will use the same `sid` value.

The `connect` handler is the place where the server can perform authentication. The value returned by this handler is used to determine if the connection is accepted or rejected. When the handler does not return any value (which is the same as returning `None`) or when it returns `True` the connection is accepted. If the handler returns `False` or any JSON compatible data type (string, integer, list or dictionary) the connection is rejected. A rejected connection triggers a response with a 401 status code.

The `data` argument passed to the `'message'` event handler contains application-specific data provided by the client with the event.

3.5 Sending Messages

The server can send a message to any client using the `send()` method:

```
eio.send(sid, {'foo': 'bar'})
```

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

```
await eio.send(sid, {'foo': 'bar'})
```

The first argument provided to the method is the connection identifier for the recipient client. The second argument is the data that is passed on to the server. The data can be of type `str`, `bytes`, `dict` or `list`. The data included inside dictionaries and lists is also constrained to these types.

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

3.6 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:

```
@eio.on('connect')
def on_connect(sid, environ):
    username = authenticate_user(environ)
    eio.save_session(sid, {'username': username})

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

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

```
@eio.on('connect')
async def on_connect(sid, environ):
    username = authenticate_user(environ)
    await eio.save_session(sid, {'username': username})

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

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

```
@eio.on('connect')
def on_connect(sid, environ):
    username = authenticate_user(environ)
    with eio.session(sid) as session:
        session['username'] = username

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

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

```
@eio.on('connect')
def on_connect(sid, environ):
    username = authenticate_user(environ)
    async with eio.session(sid) as session:
        session['username'] = username

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

Note: the contents of the user session are destroyed when the client disconnects.

3.7 Disconnecting a Client

At any time the server can disconnect a client from the server by invoking the `disconnect()` method and passing the `sid` value assigned to the client:

```
eio.disconnect(sid)
```

For the `asyncio` client this is a coroutine:

```
await eio.disconnect(sid)
```

3.8 Managing Background Tasks

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

eio.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:

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

eio.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, but the background function is.

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

```
eio.sleep(2)
```

Or for `asyncio`:

```
await eio.sleep(2)
```

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

3.9 Debugging and Troubleshooting

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

```
import engineio

# standard Python
eio = engineio.Server(logger=True)

# asyncio
eio = engineio.AsyncServer(logger=True)
```

The `logger` argument 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.

3.10 Deployment Strategies

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

3.10.1 Uvicorn, Daphne, and other ASGI servers

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

```
eio = engineio.AsyncServer(async_mode='asgi')
app = engineio.ASGIApp(eio)
```

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

```
eio = engineio.AsyncServer(async_mode='asgi')
app = engineio.ASGIApp(eio, other_app)
```

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

3.10.2 Aiohttp

aiohttp provides a framework with support for HTTP and WebSocket, based on asyncio.

Instances of class `engineio.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:

```
eio = engineio.AsyncServer(async_mode='aiohttp')
```

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

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

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

The aiohttp application is then executed in the usual manner:

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

3.10.3 Tornado

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

Instances of class `engineio.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:

```
eio = engineio.AsyncServer(async_mode='tornado')
```

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

```
app = tornado.web.Application(
    [
        (r"/engine.io/", engineio.get_tornado_handler(eio)),
    ],
    # ... other application options
)
```

The tornado application can define other routes that will coexist with the Engine.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:

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

3.10.4 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.

Instances of class `engineio.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:

```
eio = engineio.AsyncServer(async_mode='sanic')
```

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

```
app = Sanic()
eio.attach(app)
```

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

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:

```
eio = engineio.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
```

3.10.5 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 Engine.IO server deployed with eventlet has access to the long-polling and WebSocket transports.

Instances of class `engineio.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:

```
eio = engineio.Server(async_mode='eventlet')
```

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

```
app = engineio.WSGIApp(eio)
import eventlet
eventlet.wsgi.server(eventlet.listen(('', 8000)), app)
```

3.10.6 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 1` option is required. Note that a single eventlet worker can handle a large number of concurrent clients.

Another limitation when using gunicorn is that the WebSocket transport is not available, because this transport it requires extensions to the WSGI standard.

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

3.10.7 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 `engineio.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:

```
eio = engineio.Server(async_mode='gevent')
```

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

```
from gevent import pywsgi
app = engineio.WSGIApp(eio)
pywsgi.WSGIServer(('', 8000), app).serve_forever()
```

3.10.8 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 1` option is required. Note that a single gevent worker can handle a large number of concurrent clients.

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

3.10.9 uWSGI

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

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

```
# gevent with uWSGI
eio = engineio.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
```

3.10.10 Standard Threads

While not comparable to eventlet and gevent in terms of performance, the Engine.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 `engineio.Server` will automatically use the threading mode if neither eventlet nor gevent are not installed. To request the threading mode explicitly, the `async_mode` option can be given in the constructor:

```
eio = engineio.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 Engine.IO application combined with a Flask web application, using Flask's development web server based on Werkzeug:

```
eio = engineio.Server(async_mode='threading')
app = Flask(__name__)
app.wsgi_app = engineio.WSGIApp(eio, app.wsgi_app)
```

(continues on next page)

(continued from previous page)

```
# ... Engine.IO and Flask handler functions ...

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

The example that follows shows how to start an Engine.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.

3.10.11 Scalability Notes

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

- Each Engine.IO server process must be able to handle multiple requests concurrently. This is required 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 process. Load balancers call this *sticky sessions*, or *session affinity*.

3.11 Cross-Origin Controls

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

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

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

CHAPTER

FOUR

API REFERENCE

class engineio.**Client**(*logger=False*, *json=None*, *request_timeout=5*, *http_session=None*, *ssl_verify=True*, *handle_sigint=True*, *websocket_extra_options=None*)

An Engine.IO client.

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

Parameters

- **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.
- **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, custom CA bundle, 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`.
- **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.
- **websocket_extra_options** – Dictionary containing additional keyword arguments passed to `websocket.create_connection()`.

connect(*url*, *headers=None*, *transports=None*, *engineio_path='engine.io'*)

Connect to an Engine.IO server.

Parameters

- **url** – The URL of the Engine.IO server. It can include custom query string parameters if required by the server.
- **headers** – A dictionary with custom headers to send with the connection request.
- **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.

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

Example usage:

```
eio = engineio.Client()
eio.connect('http://localhost:5000')
```

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.

send(*data*)

Send a message to the server.

Parameters

data – The data to send to the server. Data can be of type `str`, `bytes`, `list` or `dict`. If a `list` or `dict`, the data will be serialized as JSON.

disconnect(*abort=False*)

Disconnect from the server.

Parameters

abort – If set to `True`, do not wait for background tasks associated with the connection to end.

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

Start a background task.

This is a utility function that applications can use to start a background task.

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()` method can be invoked to wait for the task to complete.

sleep(*seconds=0*)

Sleep for the requested amount of time.

create_queue(**args*, ***kwargs*)

Create a queue object.

create_event(**args*, ***kwargs*)

Create an event object.

on(*event*, *handler=None*)

Register an event handler.

Parameters

- **event** – The event name. Can be `'connect'`, `'message'` or `'disconnect'`.
- **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.

Example usage:

```
# as a decorator:
@eio.on('connect')
def connect_handler():
    print('Connection request')

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

transport()

Return the name of the transport currently in use.

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

class engineio.**AsyncClient**(*logger=False*, *json=None*, *request_timeout=5*, *http_session=None*, *ssl_verify=True*, *handle_sigint=True*, *websocket_extra_options=None*)

An Engine.IO client for asyncio.

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

Parameters

- **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.
- **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, custom CA bundle, 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`.
- **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.
- **websocket_extra_options** – Dictionary containing additional keyword arguments passed to `aiohttp.ws_connect()`.

async connect(*url*, *headers=None*, *transports=None*, *engineio_path='engine.io'*)

Connect to an Engine.IO server.

Parameters

- **url** – The URL of the Engine.IO server. It can include custom query string parameters if required by the server.
- **headers** – A dictionary with custom headers to send with the connection request.

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

Note: this method is a coroutine.

Example usage:

```
eio = engineio.Client()
await eio.connect('http://localhost:5000')
```

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.

async send(*data*)

Send a message to the server.

Parameters
data – The data to send to the server. Data can be of type `str`, `bytes`, `list` or `dict`. If a `list` or `dict`, the data will be serialized as JSON.

Note: this method is a coroutine.

async disconnect(*abort=False*)

Disconnect from the server.

Parameters
abort – If set to `True`, do not wait for background tasks associated with the connection to end.

Note: this method is a coroutine.

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

Start a background task.

This is a utility function that applications can use to start a background task.

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.

async sleep(*seconds=0*)

Sleep for the requested amount of time.

Note: this method is a coroutine.

create_queue()

Create a queue object.

create_event()

Create an event object.

on(*event*, *handler=None*)

Register an event handler.

Parameters

- **event** – The event name. Can be `'connect'`, `'message'` or `'disconnect'`.
- **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.

Example usage:

```
# as a decorator:
@eio.on('connect')
def connect_handler():
    print('Connection request')

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

transport()

Return the name of the transport currently in use.

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

class engineio.**Server**(*async_mode=None*, *ping_interval=25*, *ping_timeout=20*, *max_http_buffer_size=1000000*, *allow_upgrades=True*, *http_compression=True*, *compression_threshold=1024*, *cookie=None*, *cors_allowed_origins=None*, *cors_credentials=True*, *logger=False*, *json=None*, *async_handlers=True*, *monitor_clients=None*, *transports=None*, ***kwargs*)

An Engine.IO server.

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

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 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 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`.
- **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.
- **async_handlers** – If set to `True`, run message event handlers in non-blocking threads. To run handlers synchronously, set to `False`. 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`.
- **transports** – The list of allowed transports. Valid transports are `'polling'` and `'websocket'`. Defaults to `['polling', 'websocket']`.
- **kwargs** – Reserved for future extensions, any additional parameters given as keyword arguments will be silently ignored.

send(*sid*, *data*)

Send a message to a client.

Parameters

- **sid** – The session id of the recipient client.
- **data** – The data to send to the client. Data can be of type `str`, `bytes`, `list` or `dict`. If a `list` or `dict`, the data will be serialized as JSON.

send_packet(*sid*, *pkt*)

Send a raw packet to a client.

Parameters

- **sid** – The session id of the recipient client.
- **pkt** – The packet to send to the client.

get_session(*sid*)

Return the user session for a client.

Parameters
sid – The session id of the client.

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.

save_session(*sid*, *session*)

Store the user session for a client.

Parameters

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

session(*sid*)

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:

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

disconnect(*sid=None*)

Disconnect a client.

Parameters
sid – The session id of the client to close. If this parameter is not given, then all clients are closed.

handle_request(*environ*, *start_response*)

Handle an HTTP request from the client.

This is the entry point of the Engine.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.

shutdown()

Stop Socket.IO background tasks.

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

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.

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.

create_event(**args*, ***kwargs*)

Create an event object using the appropriate async model.

This is a utility function that applications can use to create an event without having to worry about using the correct call for the selected async mode.

create_queue(**args*, ***kwargs*)

Create a queue object using the appropriate async model.

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

generate_id()

Generate a unique session id.

get_queue_empty_exception()

Return the queue empty exception for the appropriate async model.

This is a utility function that applications can use to work with a queue without having to worry about using the correct call for the selected async mode.

on(*event*, *handler=None*)

Register an event handler.

Parameters

- **event** – The event name. Can be `'connect'`, `'message'` or `'disconnect'`.
- **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.

Example usage:

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

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

The handler function receives the `sid` (session ID) for the client as first argument. The `'connect'` event handler receives the WSGI environment as a second argument, and can return `False` to reject the connection. The `'message'` handler receives the message payload as a second argument. The `'disconnect'` handler does not take a second argument.

transport(*sid*)

Return the name of the transport used by the client.

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

Parameters
sid – The session of the client.

class engineio.**AsyncServer**(*async_mode=None*, *ping_interval=25*, *ping_timeout=20*, *max_http_buffer_size=1000000*, *allow_upgrades=True*, *http_compression=True*, *compression_threshold=1024*, *cookie=None*, *cors_allowed_origins=None*, *cors_credentials=True*, *logger=False*, *json=None*, *async_handlers=True*, *monitor_clients=None*, *transports=None*, ***kwargs*)

An Engine.IO server for asyncio.

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

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.
- **http_compression** – Whether to compress packages when using the polling transport.

- **compression_threshold** – Only compress messages when their byte size is greater than this value.
- **cookie** – 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.
- **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`, run message event handlers in non-blocking threads. To run handlers synchronously, set to `False`. 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`.
- **transports** – The list of allowed transports. Valid transports are `'polling'` and `'websocket'`. Defaults to `['polling', 'websocket']`.
- **kwargs** – Reserved for future extensions, any additional parameters given as keyword arguments will be silently ignored.

attach(*app*, *engineio_path='engine.io'*)

Attach the Engine.IO server to an application.

async send(*sid*, *data*)

Send a message to a client.

Parameters

- **sid** – The session id of the recipient client.
- **data** – The data to send to the client. Data can be of type `str`, `bytes`, `list` or `dict`. If a `list` or `dict`, the data will be serialized as JSON.

Note: this method is a coroutine.

async send_packet(*sid*, *pkt*)

Send a raw packet to a client.

Parameters

- **sid** – The session id of the recipient client.
- **pkt** – The packet to send to the client.

Note: this method is a coroutine.

async get_session(*sid*)

Return the user session for a client.

Parameters
sid – The session id of the client.

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 save_session(*sid*, *session*)

Store the user session for a client.

Parameters

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

session(*sid*)

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:

```
@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 disconnect(*sid=None*)

Disconnect a client.

Parameters
sid – The session id of the client to close. If this parameter is not given, then all clients are closed.

Note: this method is a coroutine.

async handle_request(**args*, ***kwargs*)

Handle an HTTP request from the client.

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

Note: this method is a coroutine.

async shutdown()

Stop Socket.IO background tasks.

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

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.

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.

create_queue(**args*, ***kwargs*)

Create a queue object using the appropriate async model.

This is a utility function that applications can use to create a queue without having to worry about using the correct call for the selected async mode. For asyncio based async modes, this returns an instance of `asyncio.Queue`.

get_queue_empty_exception()

Return the queue empty exception for the appropriate async model.

This is a utility function that applications can use to work with a queue without having to worry about using the correct call for the selected async mode. For asyncio based async modes, this returns an instance of `asyncio.QueueEmpty`.

create_event(**args*, ***kwargs*)

Create an event object using the appropriate async model.

This is a utility function that applications can use to create an event without having to worry about using the correct call for the selected async mode. For asyncio based async modes, this returns an instance of `asyncio.Event`.

generate_id()

Generate a unique session id.

on(*event*, *handler=None*)

Register an event handler.

Parameters

- **event** – The event name. Can be `'connect'`, `'message'` or `'disconnect'`.
- **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.

Example usage:

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

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

The handler function receives the `sid` (session ID) for the client as first argument. The `'connect'` event handler receives the WSGI environment as a second argument, and can return `False` to reject the connection. The `'message'` handler receives the message payload as a second argument. The `'disconnect'` handler does not take a second argument.

transport(*sid*)

Return the name of the transport used by the client.

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

Parameters
sid – The session of the client.

class engineio.**WSGIApp**(*engineio_app*, *wsgi_app=None*, *static_files=None*, *engineio_path='engine.io'*)

WSGI application middleware for Engine.IO.

This middleware dispatches traffic to an Engine.IO application. It can also serve a list of static files to the client, or forward unrelated HTTP traffic to another WSGI application.

Parameters

- **engineio_app** – The Engine.IO server. Must be an instance of the `engineio.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.
- **engineio_path** – The endpoint where the Engine.IO application should be installed. The default value is appropriate for most cases.

Example usage:

```
import engineio
import eventlet

eio = engineio.Server()
app = engineio.WSGIApp(eio, static_files={
    '/': {'content_type': 'text/html', 'filename': 'index.html'},
    '/index.html': {'content_type': 'text/html',
                    'filename': 'index.html'},
})
eventlet.wsgi.server(eventlet.listen(('', 8000)), app)
```

class engineio.**ASGIApp**(*engineio_server*, *other_asgi_app=None*, *static_files=None*, *engineio_path='engine.io'*, *on_startup=None*, *on_shutdown=None*)

ASGI application middleware for Engine.IO.

This middleware dispatches traffic to an Engine.IO application. It can also serve a list of static files to the client, or forward unrelated HTTP traffic to another ASGI application.

Parameters

- **engineio_server** – The Engine.IO server. Must be an instance of the `engineio.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.
- **engineio_path** – The endpoint where the Engine.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 Engine.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:

```
import engineio
import uvicorn

eio = engineio.AsyncServer()
app = engineio.ASGIApp(eio, static_files={
    '/': {'content_type': 'text/html', 'filename': 'index.html'},
    '/index.html': {'content_type': 'text/html',
                    'filename': 'index.html'},
})
uvicorn.run(app, '127.0.0.1', 5000)
```

async **not_found**(*receive*, *send*)

Return a 404 Not Found error to the client.

class engineio.**Middleware**(*engineio_app*, *wsgi_app=None*, *engineio_path='engine.io'*)

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

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

T

W