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.

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]"
```

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.

Creating a Client Instance

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

```
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:

```
import socketio

# standard Python
sio = socketio.SimpleClient()

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

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')
```

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:

```
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:

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

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.

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)
```

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()
```

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.

Using the Event-Driven Client

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()
```

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:

```
@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:

```
@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.

Catch-All Event 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:

```
@sio.on('*')
def catch_all(event, data):
    pass
```

Asyncio clients can also use a coroutine:

```
@sio.on('*')
async def catch_all(event, 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.

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.

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')
```

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.

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.

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:

```
@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:

```
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.

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`**:

```
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:

```
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'))
```

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.

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()
```

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:

```
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:

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

Or for `asyncio`:

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

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

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.