Getting Started

Initialization

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

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

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

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

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

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

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

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

Receiving Messages

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

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

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

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

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

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

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

Custom named events can also support multiple arguments:

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

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

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

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

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

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

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

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

```
def my_function_handler(data):
    pass

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

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

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

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

Sending Messages

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

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

```
from flask_socketio import send, emit

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

@socketio.on('json')
def handle_json(json):
```

```
        send(json, json=True)

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

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

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

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

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

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

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

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

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

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

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

Broadcasting

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

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

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

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

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

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

Rooms

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

```
from flask_socketio import join_room, leave_room

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

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

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

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

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

Connection Events

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

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

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

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

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

```
from flask_socketio import ConnectionRefusedError

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

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

Class-Based Namespaces

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

```
from flask_socketio import Namespace, emit

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

    def on_disconnect(self):
        pass

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

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

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

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

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

Error Handling

Flask-SocketIO can also deal with exceptions:

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

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

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

Error handler functions take the exception object as an argument.

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

```
from flask import request

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

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

Debugging and Troubleshooting

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

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

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

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