Open-Source Report

Proof of knowing your stuff in CSE312

Guidelines

Provided below is a template you must use to write your reports for your project.

Here are some things to note when working on your report, specifically about the **General Information & Licensing** section for each technology.

- **Code Repository**: Please link the code and not the documentation. If you'd like to refer to the documentation in the **Magic** section, you're more than welcome to, but we need to see the code you're referring to as well.
- **License Type**: Three letter acronym is fine.
- **License Description**: No need for the entire license here, just what separates it from the rest.
- **License Restrictions**: What can you *not* do as a result of using this technology in your project? Some licenses prevent you from using the project for commercial use, for example.

Also, feel free to extend the cell of any section if you feel you need more room.

If there's anything we can clarify, please don't hesitate to reach out! You can reach us using the methods outlined on the course website or see us during our office hours.

[Flask SocketIO]

General Information & Licensing

Code Repository	https://github.com/miguelgrinberg/Flask-SocketIO
License Type	MIT
License Description	● Gives users the permission to reuse code for any purpose. Is under the BSD-Style and permissive class of licenses. ● Doesn't require modifications to be open sourced ● Only condition required to use software is to include the same copyright notice in all copies or any substantial portions of the software
License Restrictions	● If the code is distributed, the license automatically becomes a GPL license. ● There are no warranties for the software and no attribution ● No patent protection and no copyleft

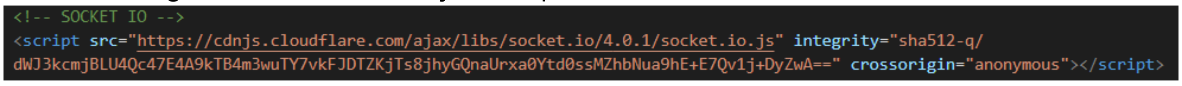
Magic

- How does this technology do what it does?

The implementation of the websocket used in our application was through using Flask-SocketIO for server implementation and Socket.IO for client implementation. SocketIO is a JavaScript library that creates the communication protocol that builds on top of HTTP and websocket, providing extra features such as automatic reconnection, event based notifications, etc. Flask-SocketIO is an implementation of SocketIO server side protocol as a Flask extension

- Socket.IO for client implementation

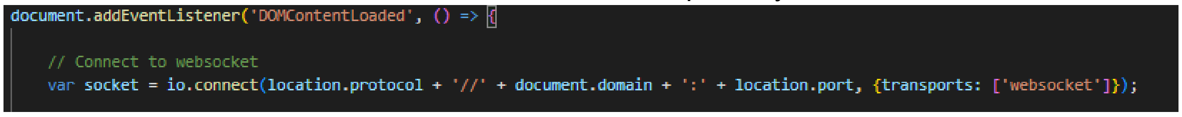
We start off with implementing an instance of the socket on the client side by CDN sourcing it on the client side javascript file. In our code that would look like:



```
<!-- SOCKET IO -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.0.1/socket.io.js" integrity="sha512-q/
dWJ3kcmjBLU4Qc47E4A9kTB4m3wuTY7vkFJDTZKjTs8jhyGQnaUrxa0Ytd0ssMZhbNua9hE+E7Qv1j+DyZwA==" crossorigin="anonymous"></script>
```

Link to our code:
https://github.com/vwong175/cse312_group/blob/7dd779930ab091e017147d4959863fd15a24e6bf/templates/layout.html#L16

After the DOM finishes loading, the instance of the socket is created on the client-side, which is shown in the below lines of code from our repository.



```
document.addEventListener('DOMContentLoaded', () => {

    // Connect to websocket
    var socket = io.connect(location.protocol + '//' + document.domain + ':' + location.port, {transports: ['websocket']});
```

The above line of code demonstrates the similarity to setting up the websocket connection as demonstrated in HW3 when writing

```
const socket = new WebSocket('ws://' + window.location.host + '/websocket');
```

Because the io method is bound to global scope in the standalone build, upon calling io.connect(), we are establishing the connection as provided with the CDN sourced socketio and passing in the URI of our page.

The lifecycle of the websocket connection is that establishing the websocket connection requires the manager to connect or reconnect to the server. The socket sends a handshake packet and will try to establish a connection. For the handshake, it happens at the beginning of the Engine.IO connection where it sends data like:

```
{
  "sid": "FSDjX-WRwSA4zTZMALqx",
  "upgrades": ["websocket"],
  "pingInterval": 25000,
  "pingTimeout": 20000
}
```

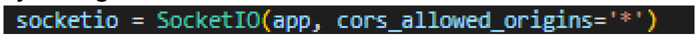
Where the sid is the id of the session, the upgrades array contains the list of all “better” transports that are supported by the server, and the pingInterval and pingTimeout values are used in the heartbeat mechanism of the socket connection.

If it can’t establish a connection, i.e connect_error, then it will disconnect. If it could connect, then the socket is connected and upon disconnection the socket disconnects and the manager is closed. The manager will try to connect back if possible.

In our code, we specify that we want the only transports for engineio of the socket to only do the websocket transport rather than long polling. We do this based on the project requirements of disabling long polling.

- Flask-socketIO for server implementation

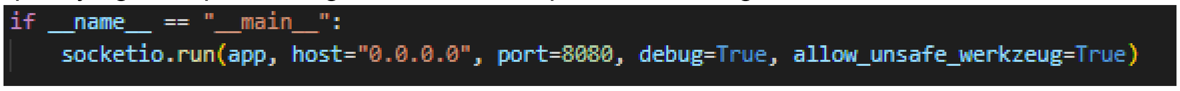
And similarly we will need to set up the socket on the server side which is done by importing SocketIO from flask_socketio, creating an instance of the server side socket io by doing



```python
socketio = SocketIO(app, cors_allowed_origins='*')
```

To create an SocketIO object, (https://github.com/miguelgrinberg/Flask-SocketIO/blob/91b5ddc31bebeb6241d281252c711b160550ce01/src/flask_socketio/__init__.py#L54), pass in is the flask application instance. If the application instance isn't known at the time this class is instantiated, then call socketio.init_app(app) once the application instance is available. When instantiating this object, as it can be seen in the __init__ function https://github.com/miguelgrinberg/Flask-SocketIO/blob/91b5ddc31bebeb6241d281252c711b160550ce01/src/flask_socketio/__init__.py#L171, we assign the server, the server_options, wsgi_server, handlers, and the other state variables of the server side object. If we pass our instance of the Flask application to the SocketIO instance, then it calls init_app from the app factory function, otherwise it doesn't and updates the server_options.

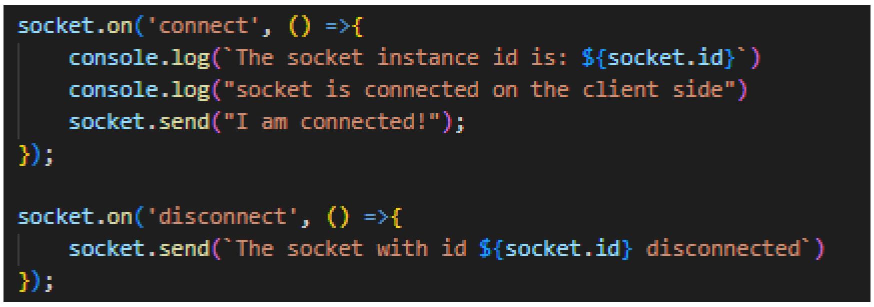
And with a variable named socketio that is assigned to the instance of the server side SocketIO, we pass our instance of the Flask application to socketio.run() while also specifying the optional arguments of host, port, and debug.



```python
if __name__ == "__main__":
    socketio.run(app, host="0.0.0.0", port=8080, debug=True, allow_unsafe_werkzeug=True)
```

The handlers are registered for these events with a similar use case of how routes are handled by the view function for the application routes. The extension `if __name__ == '__main__'`) is initialized in the usual way but to simplify up the server a custom run() method is used instead of flask run or app.run(). The method starts the eventlet or gevent if they are installed. It uses gunicorn with the eventlet or gevent workers should work. To receive a websocket message from the client, the application defines an event handler using the socketio.on decorator.

On the client side, we are able to implement functionality to send/emit messages through the socket to the server, and when we get a websocket response from the server, on the client side we are also able to handle those events being sent from the server. This is done with creating event handlers and an example of two predefined events in SocketIO would be the two below:

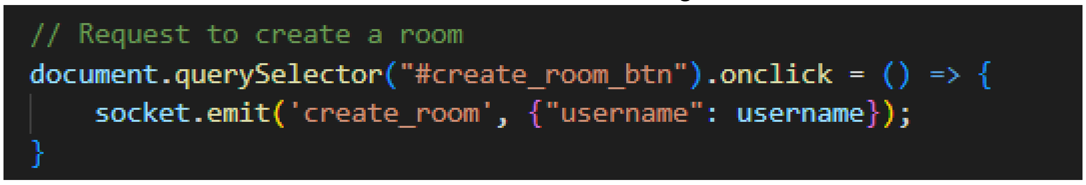


```javascript
socket.on('connect', () =>{
    console.log(`The socket instance id is: ${socket.id}`)
    console.log("socket is connected on the client side")
    socket.send("I am connected!");
});

socket.on('disconnect', () =>{
    socket.send(`The socket with id ${socket.id} disconnected`)
});
```

The first argument to the decorator is the event name and the second argument is the function that is executed when on the client side it receives a websocket event response on that event name from the server.

A more concrete example of sending a websocket message on the client side is when we are able to create a room such that when the user interacts with the create room button, our client side instance of socketio will emit a message to the server side.

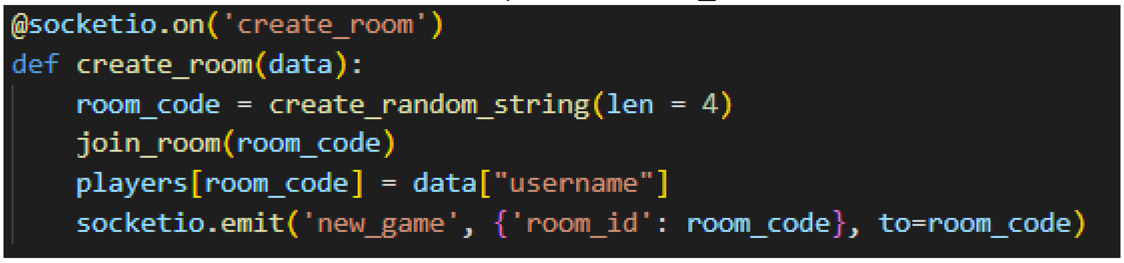


```
// Request to create a room
document.querySelector("#create_room_btn").onclick = () => {
    socket.emit('create_room', {"username": username});
}
```

On the client side, not only are we able to listen to events happening, but we are also able to send messages with the socket.send() or socket.emit() functions. With socket.send(), we are able to send a standard message, i.e the predefined “message” event handler that the server can listen for, and with socket.emit(), we can send messages under a custom application defined event name.

On the server side, as stated previously, will have an instance of FlaskSocketIO as the socket, to which it also uses event handlers to listen to events that will be sent/emitted from the client side of the websocket.

Here is an example of an event handler on the server side where the server will execute the below code whenever it hears a request for create_room.



```
@socketio.on('create_room')
def create_room(data):
    room_code = create_random_string(len = 4)
    join_room(room_code)
    players[room_code] = data["username"]
    socketio.emit('new_game', {'room_id': room_code}, to=room_code)
```

Inspecting the event handler of .on() with the link to the source code of the definition of .on(): https://github.com/miguelgrinberg/Flask-SocketIO/blob/91b5ddc31bebeb6241d281252c711b160550ce01/src/flask_socketio/__init__.py#L258, within the on function definition is the definition for the decorator and it returns the handle_event. As can be seen within the function body, the namespace, if given, will take upon the inputted namespace or default to ‘/’, the root path if it is not given. The namespace within SocketIO is the equivalent to a path.

Upon running our application and going through the stack trace with the debugger, our app directs a logged in user to our lobby page where they have the ability to either create a room and essentially become the host of a new game with a new player, or they can choose to enter a known and valid room code and join a game that another player has made.

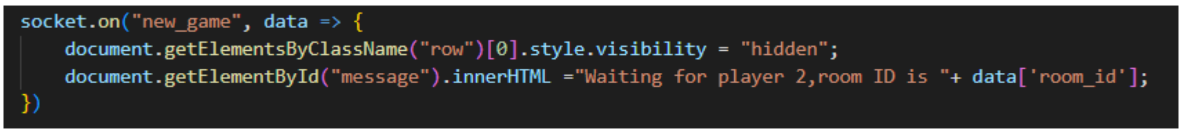
Starting with the create_room socket event listener as shown previously, and placing a breakpoint at join_room(room_code), stepping into this function definition (https://github.com/miguelgrinberg/Flask-SocketIO/blob/91b5ddc31bebeb6241d281252c711b160550ce01/src/flask_socketio/__init__.py#L1012) we can see that the purpose of this function is to put a user in the room that was created, or in other words, the room string that we provided as input. It puts the user in the room under the current namespace and the user and the namespace are obtained from the event context. The required parameter is a string that represents the name of the room to join and an optional parameter is the session id of the client. If it is not provided, the client is obtained from the request context. The other optional parameter is the namespace which represents the namespace for the room, if it is also not provided, the namespace is obtained from the request context.

The first three lines of join_room are assigning values to the variables socketio, sid, and namespace. The function ends with calling `socketio.server.enter_room(sid, room, namespace=namespace)` to which what it does is uses `self.manager.enter_room(sid, namespace, room)` (https://github.com/miguelgrinberg/python-socketio/blob/1fa70ccefa3e4447c5be42cae8d7222a57eaeb4c/src/socketio/server.py#L422) and essentially is the part where it adds a client to a room. When entering the manager.enter_room() function definition (https://github.com/miguelgrinberg/python-socketio/blob/1fa70ccefa3e4447c5be42cae8d7222a57eaeb4c/src/socketio/base_manager.py#L116) this is the part where it assigns the eio_sid if given to the room's namespace, room, and sid in the line:
`self.rooms[namespace][room][sid] = eio_sid`

Continuing with the trace of the program, the debugger brings us the next important line which in our code is: `socketio.emit('new_game', {'room_id': room_code}, to=room_code).` We do this in our code because after a user joins a room, we want the socket to emit a message back to the client side, an event message named 'new_game' as well as the JSON data we are sending. In this case we are sending the data payload dictionary with a key named 'room_id' with a value of room_code. With emitting messages, we have the ability to emit this message only to a specific room, which is specified in the third argument with `to=room_code`. Upon stepping into the `socketio.emit()` function (https://github.com/miguelgrinberg/Flask-SocketIO/blob/91b5ddc31bebeb6241d281252c711b160550ce01/src/flask_socketio/__init__.py#L401), in a general overview what this function does is emit a SocketIO event to one or more connected clients with a JSON blob that can be attached to the event as the payload. It assigns the necessary variables like the namespace, to, include_self, and callback. If there is a callback function that was provided, then it wraps the callback so that it sets the app and request contexts, but in the context of our specific example of joining a room, we don't provide a callback function. Thus this brings our debugger to the line https://github.com/miguelgrinberg/Flask-SocketIO/blob/91b5ddc31bebeb6241d281252c711b160550ce01/src/flask_socketio/__init__.py#L462 (line 462): where it will call the `self.server.emit(event)` function by passing the event we defined to self.server. By calling server.emit(event) which is defined here: https://github.com/csulliv9/python_socketio/blob/1600aa424c334eb9328dc8ae633ada68f65c3923/socketio/server.py#L177 , which will call `self.manager.emit()` (https://github.com/csulliv9/python_socketio/blob/1600aa424c334eb9328dc8ae633ada68f65c3923/socketio/base_manager.py#L115) with the same arguments. This chain of function calls is to emit a message to either a single client, a room, or all the clients connected to the namespace. The last emit function call is calling `self.server._emit_internal()` (https://github.com/csulliv9/python_socketio/blob/1600aa424c334eb9328dc8ae633ada68f65c3923/socketio/server.py#L379) which will send the packet to the sid provided, namespace if provided, data, id and binary. What _send_packet does essentially is to use `self.eio.send()` to send the encoded packet to the sid.

The debugger brings us to the `_handle_event` function defined here: https://github.com/miguelgrinberg/Flask-SocketIO/blob/91b5ddc31bebeb6241d281252c711b160550ce01/src/flask_socketio/__init__.py#L789 and essentially why our code goes here is to be able to return the value of the handler().

Still using the example of the joining _game event handler, we can go back to the client side. After the server has emitted the message, specifically what we named `new_game`, on the client side, we have functionality that changes the current html page that the host user is on, to show the message that they are waiting for the other user to join the game.



```
socket.on("new_game", data => {
    document.getElementsByClassName("row")[0].style.visibility = "hidden";
    document.getElementById("message").innerHTML ="Waiting for player 2,room ID is "+ data['room_id'];
})
```

Websocket Frame Parsing:
Since Flask-SocketIO is an instance of SocketIO, how SocketIO does the frame parsing is through their socketio-parser: https://github.com/socketio/socket.io-parser . And because the socketio-parser is an instance of socket.io-protocol, the socket.IO protocol adds another layer to the Engine.IO protocol by providing extra features like multiplexing, and acknowledgement of packets. Multiplexing is synonymous with the idea of Namespaces. A namespace is a communication channel that allows you to split the logic of an application over a single shared connection. What is required to encode and decode a packet is a parser, an encoder, and a packet. For the parser, an example would be calling this line of code: `require('socket.io-parser')`. For the encoder, we used the parser and created an instance of the parser's encoder by doing something like `new parser.Encoder()`. For the packet, it will be an object that holds the type, data, and id.