Technology used:

Django Channels
https://channels.readthedocs.io/en/latest/

What it accomplishes:

A. Allow Django to use Websockets
B. Allow the server to be run asynchronously

How it accomplishes the above:

A. First, we set a key in our *settings.py* in our main folder, setting a key called ASGI_APPLICATION equal to a variable in our *routing.py*.
    - The variable is an object of class *ProtocolTypeRouter*, in Django Channel's *routing.py* which takes protocol type names to other Application instances, and dispatches the right one based on the name. In our case, we call it with the name "websocket". To map websocket connections to the specific application that we want.
        - Within the constructor of our *ProtocolTypeRouter* call, our "websocket" key is mapped to an *AuthMiddlewareStack* call, which adds keys to a dictionary called *scope*, containing information on the path, cookies, sessions, and other information derived from the constructor, facilitated through the classes of *AuthMiddleware*, *SessionMiddleware*, and *CookieMiddleware*. *AuthmiddlewareStack* and *AuthMiddleware* are located in *auth.py*, and *SessionMiddleware* and *CookieMiddleware* are located in *sessions.py*.
        - Within our *AuthMiddlewareStack* call, we input a constructor in the form of an object of class *URLRouter*, which routes requests to different applications or consumer (will elaborate what consumers are further below) based on the path found in the constructor. This function can be found in *routing.py*.
        - Within our constructor for *URLRouter*, we link the information on path routing to each app's respective *routing.py* folder. We use Django's *re_path* function to use regular expressions to identify the path that is found in the request, and to specify which function that path should call. This function can be found in Django's *urls/conf.py*.
    - In our respective apps' *consumers.py*, we initialize Consumer objects, which are created for each Websocket connection. These consumers facilitate the sending and receiving of Websocket frames with their respective client. In our project, we use Django Channels' *WebSocketConsumer* class, found in

generic/websocket.py, which gives the ability to handle Websocket connections, disconnections, parsing frames received, and sending frames. For each consumer, we put them in a dictionary to identify which consumer sent what.

B. To run the server asynchronously, Django Channels uses Daphne to run Django on an ASGI server, which allows asynchronous functionality. Django Channels runs Daphne through Channels' *Command* class in their *runserver.py*. It calls Daphne's *build_endpoint_description_strings* function in Daphne's *endpoints.py*, which builds a list of endpoint strings for the server to listen on. This is passed into the constructor of Daphne's *Server*, information located in *server.py*, to create the server for our program to run on.
    - The Daphne server uses Twisted to create the event loop to allow the server to run asynchronously. This is done through function calls from Twisted's *reactor.py*, which calls *install* from Twisted's *default.py*.

Licenses / TOS:

The License of Django Channels needs to meet three criteria:

1. Redistributions of source code must maintain the copyright notice.
2. Redistributions in binary must maintain the copyright notice in documentation.
3. Neither the name Django nor its contributors may be used to endorse or promote products derived from the software.

For our project, all three of the above are met and will be maintained.

License found here.