COS60016: Programming for Development

Assignment 2: Build a chatbot

Student: Alexandra Bain
Student Email: 103083826
Student ID: 103083826@student.swin.edu.au

Chapters

1. *Analysis of chatbot design*

Chatbot Creation

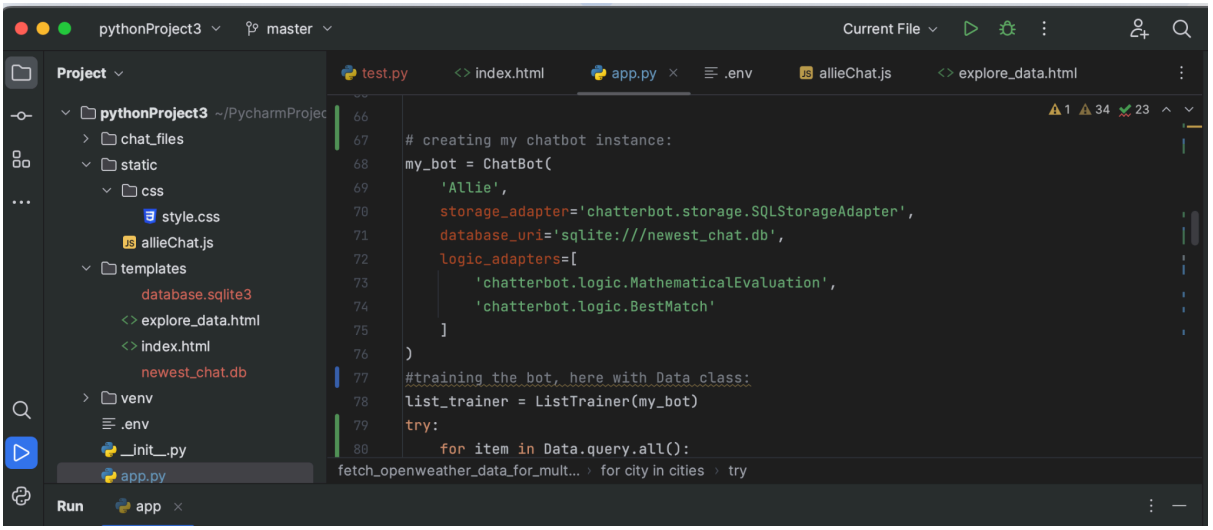
I will be building a chatbot able to integrate with the original weather application I built for the "Go Travel!" website. I set up a Flask app to pull data, with a route for displaying chatbot responses to train the chatbot.

Chatterbot and Flask App Configuration

To start with building the weather chatbot for the blogger's website, I created a fresh Python project after downloading and downgrading to python 3.7.9 to download the Chatterbot, Flask and SQLAlchemy libraries. I experienced a few compatibility issues with libraries and striking the right balance/ researching what was needed. For example, I had some issues with Spacy, which downloaded automatically with ChatterBot and was affecting other dependencies on my app. I managed to work around this by downloading ChatterBot and re-installing Stacy and then re-wrote the tagging.py non-project file (that kept being flagged as an error) without any reference to Spacy.

I initialise the app with Socket.io library, a JS library that enables real-time, bidirectional communication between web clients and servers, to keep the chatbot's weather information responses up-to-date.

I initialised my Chatterbot chatbot and named it AllieChat:

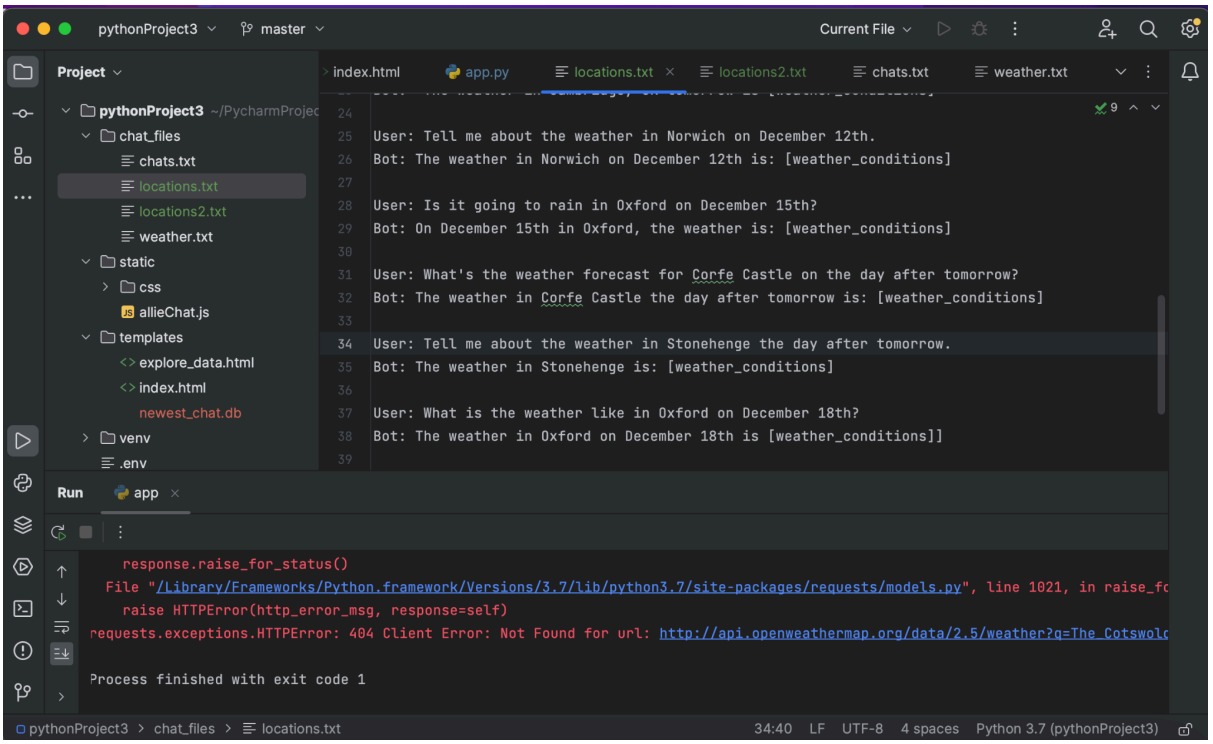


I added an index.html template page in the 'templates' directory, and a static directory with an "allieChat.js" Javascript, which implemented chat functionality.

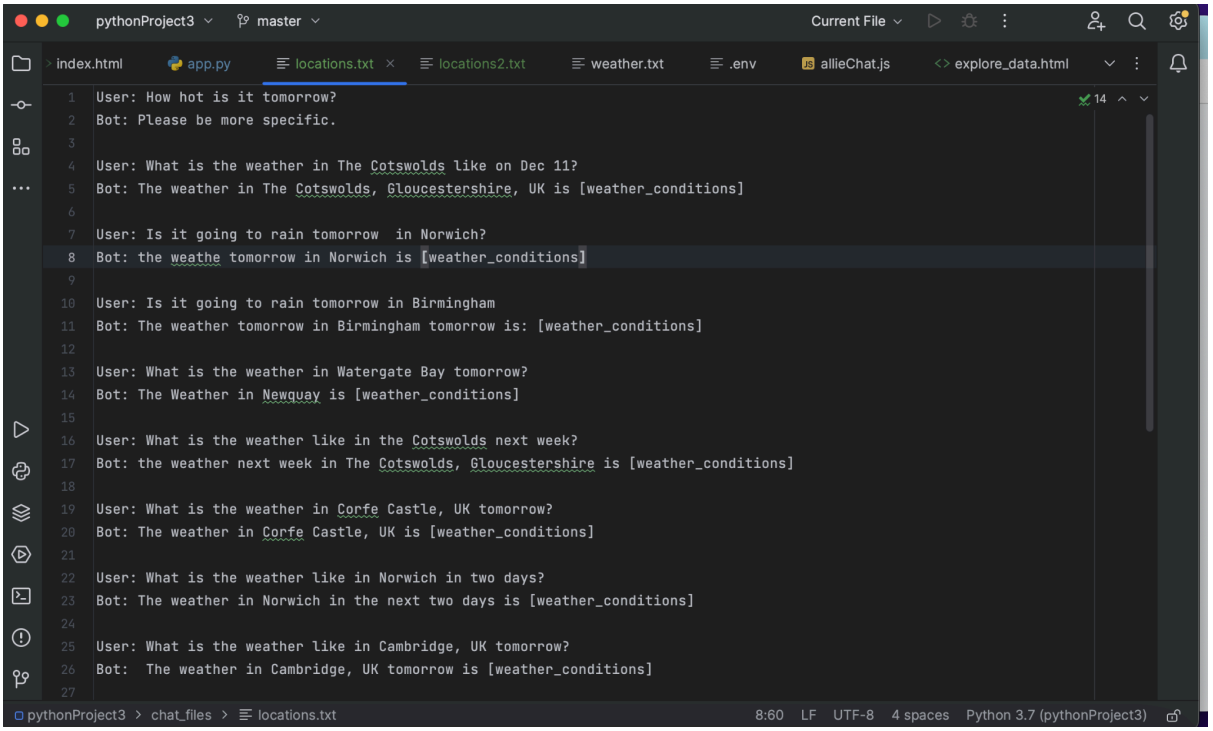
Chatbot Training

I trained the chatbot with conversation and general weather information to prepare for importing database info.

I included examples of user queries and correct responses to improve the chatbot's ability to handle a wider range of inputs. I trained with user queries that the chatbot may not handle well and got as specific as possible. I also linked it to the function weather_queries in app.py:

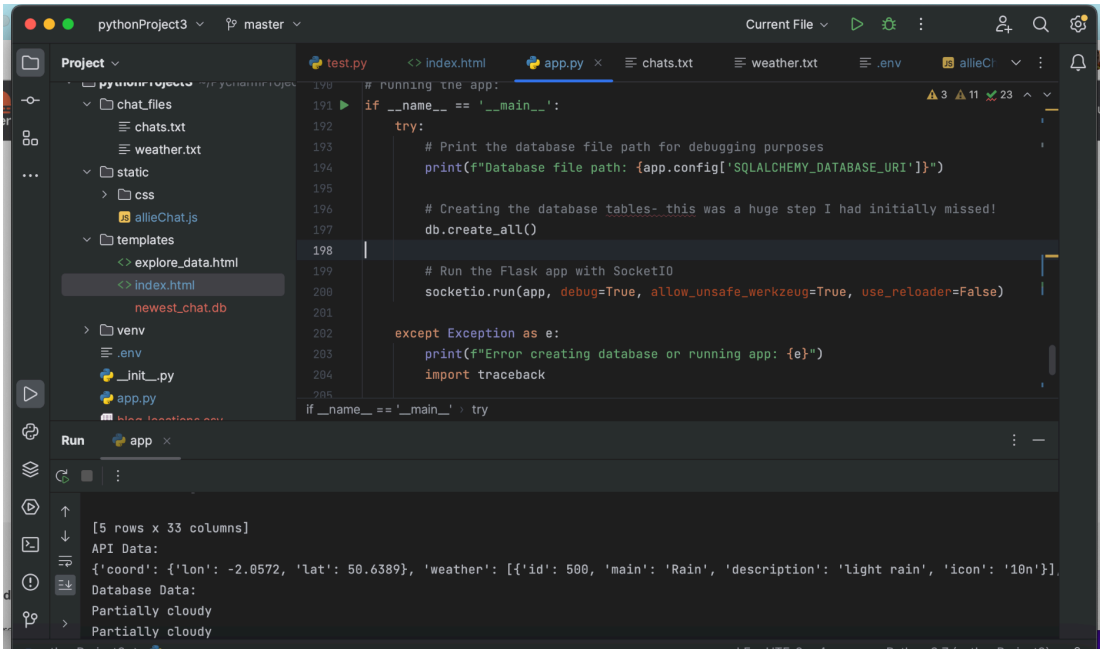


I also had to account for the fact that my API in app.py was *not* pulling information directly from The Cotswolds, but instead, Gloucestershire and instead of Watergate Bay, Newquay. Therefore, I was sure to include these locations in the text chats:

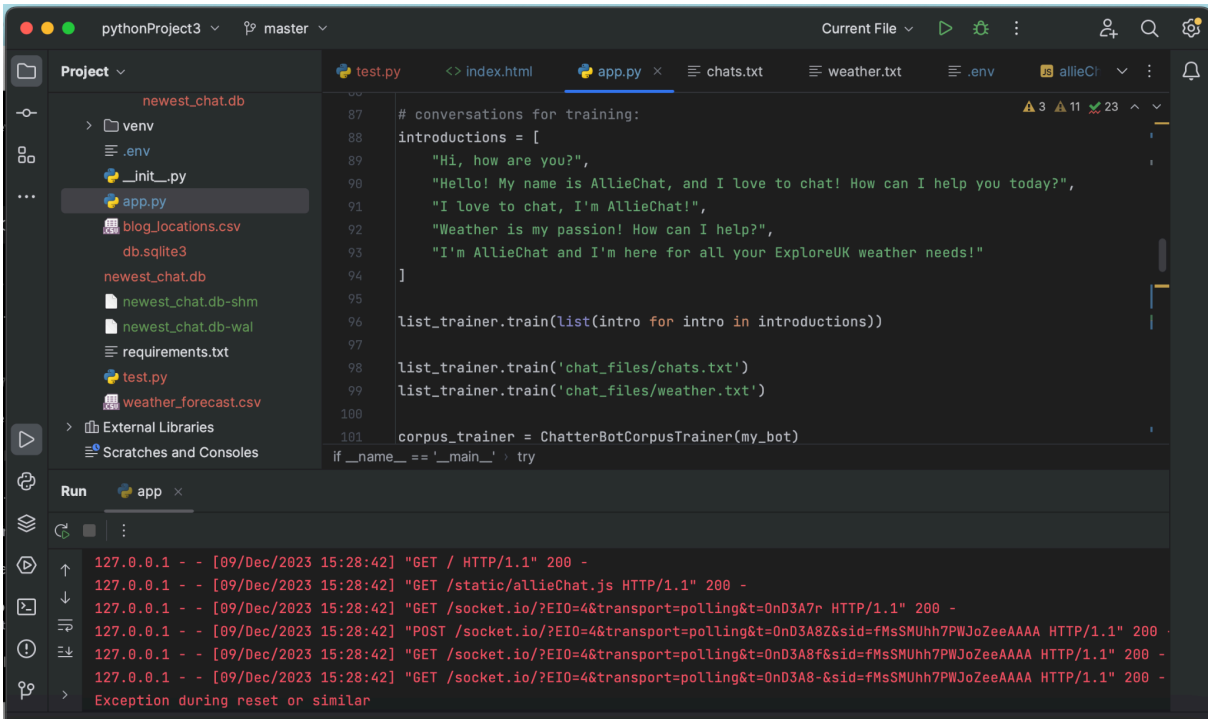


I implemented A/B Testing:

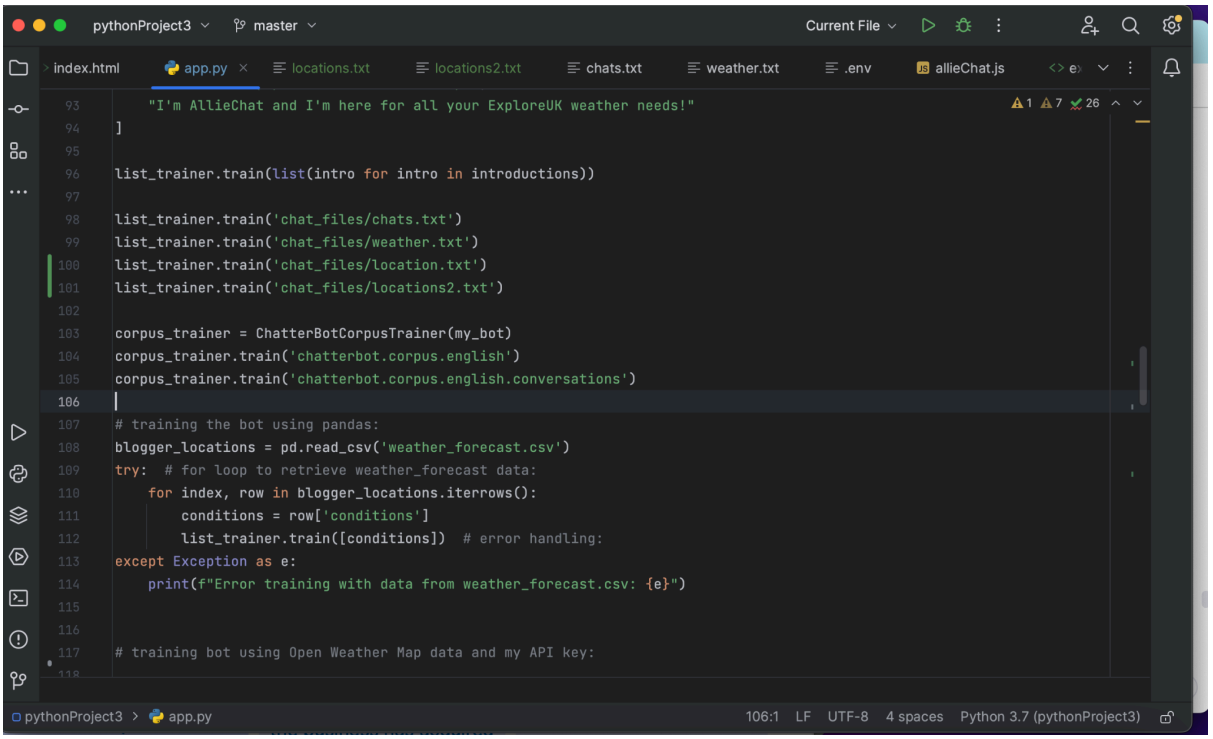
- I built multiple chatbot versions across two different app structures, using Python Flask. Experimenting with these different versions and the varying output of the chatbots, I was able to see which performed better, and;
- I tested variations in language and response structure.



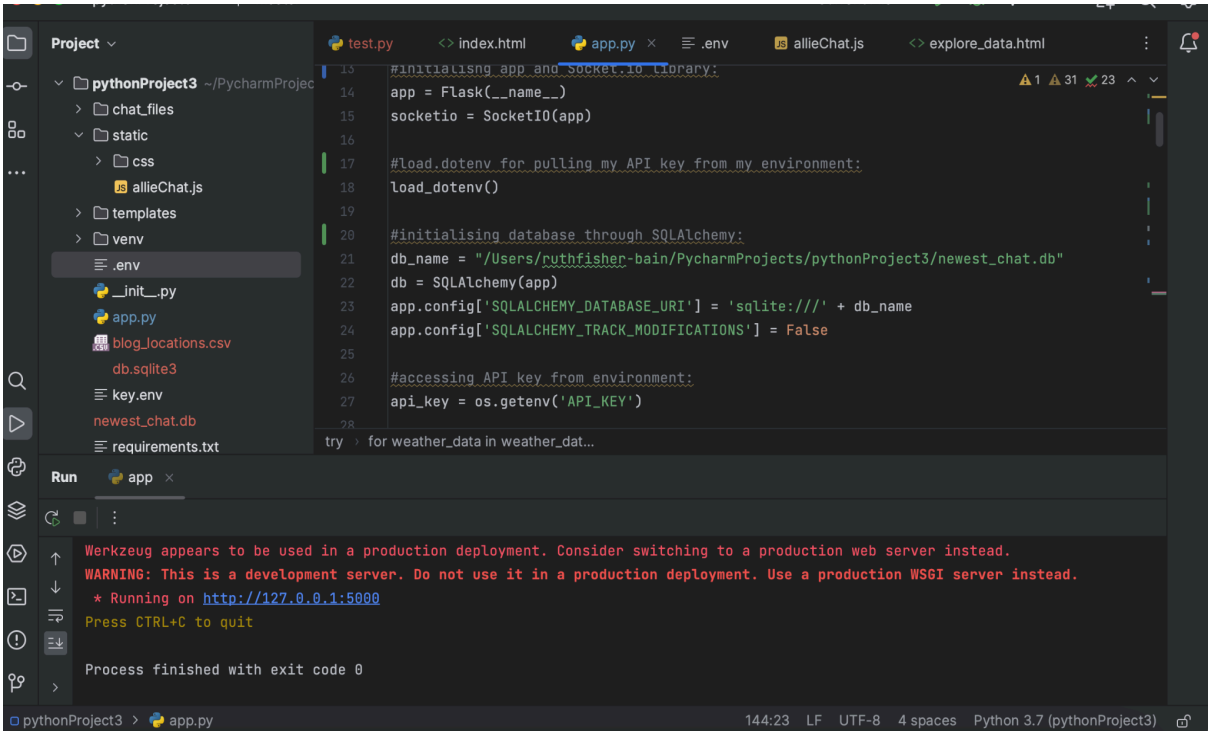
I added in ChatterBot Corpus English and Conversation trainers, and some simple AllieChat greetings, “introductions”, in a list to loop through:



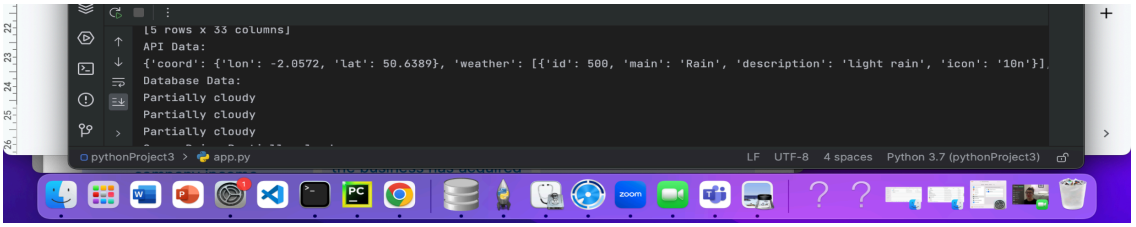
I added another two text files of example chatbot chat to chat_files:

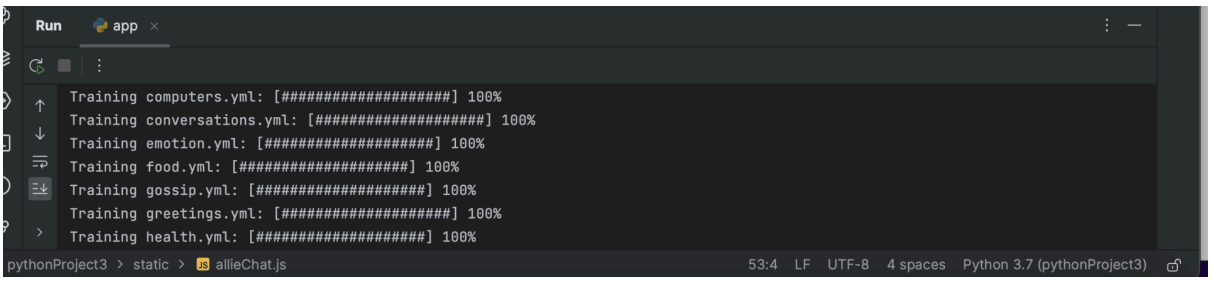


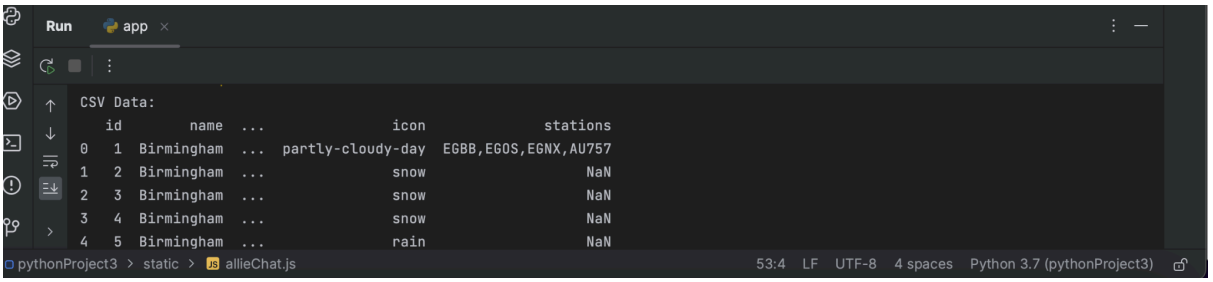
I configured ChatterBot with an SQLite storage adapter using the my_bot instance, and after building a database (see "**Wrangling Data Using SQLite Database**", page 18) I initialised the database through SQLAlchemy for Flask:

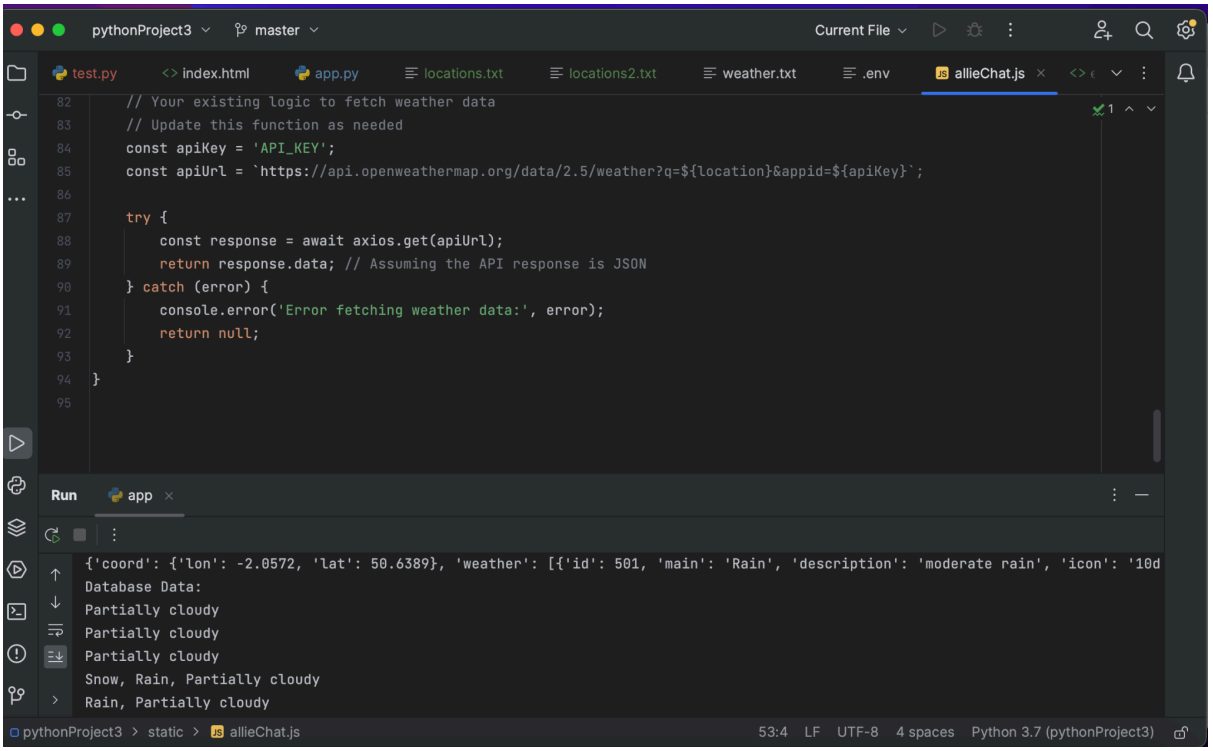


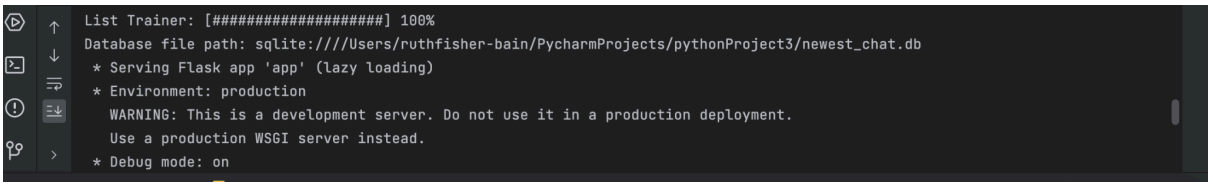
I confirmed the data was correctly being pulled from the SQL and other locations thanks to my js:







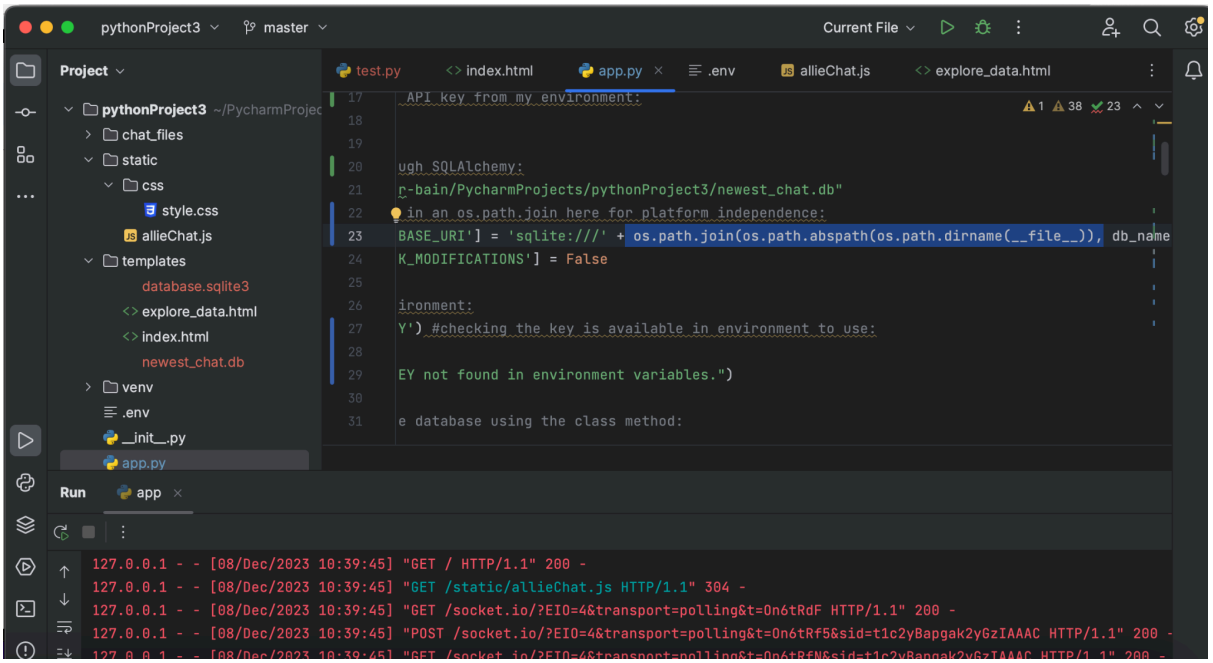




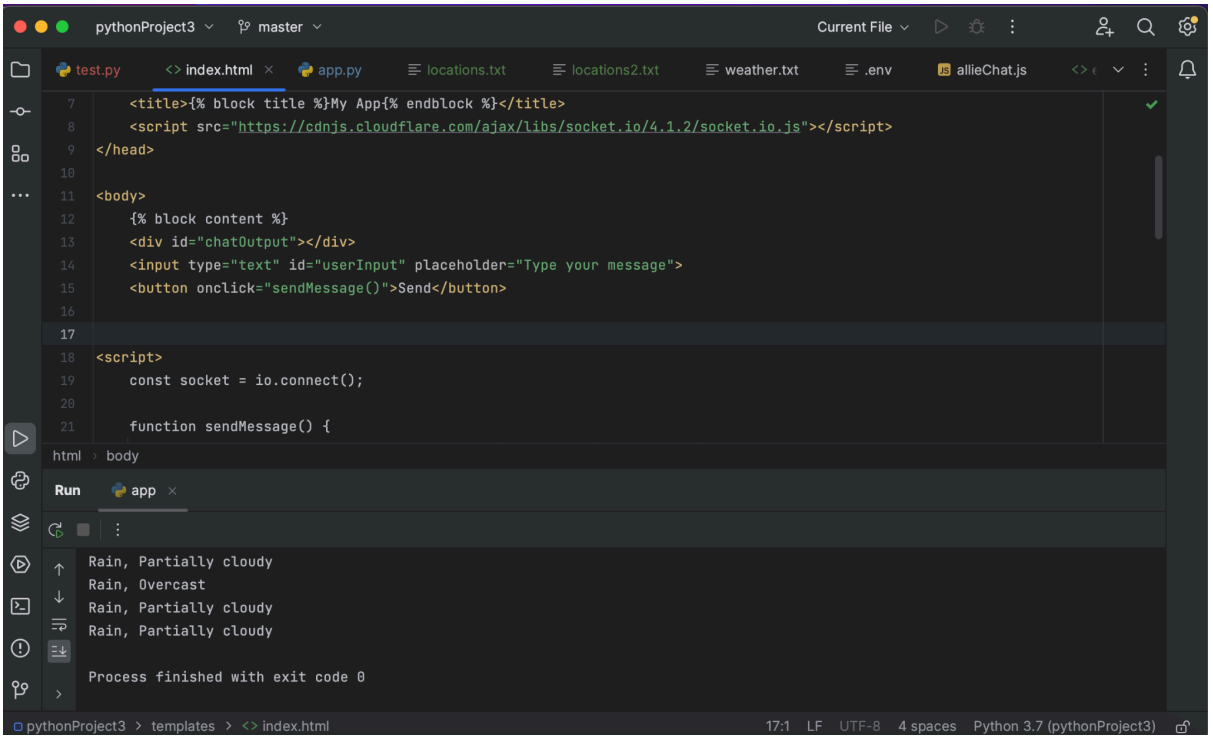
2. *Python code to build the chatbot*

Building the Bot App

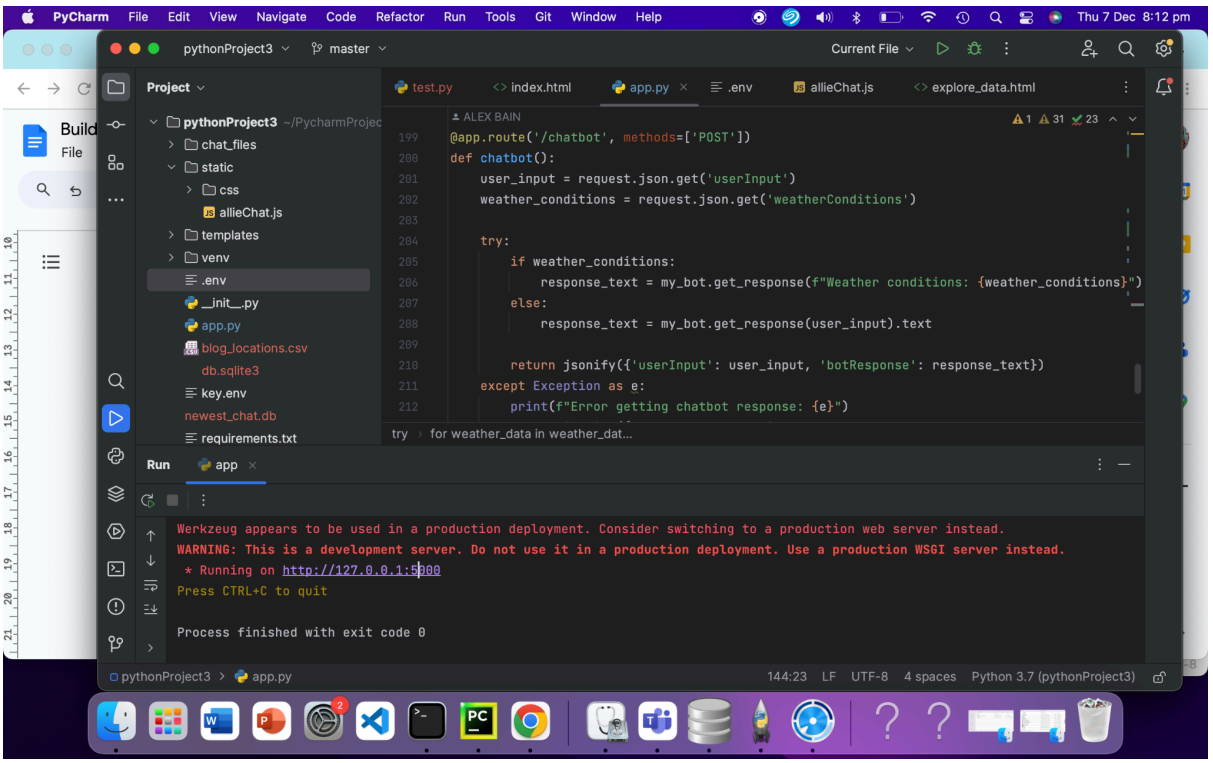
First, I added in an os.path.join function for platform independence:



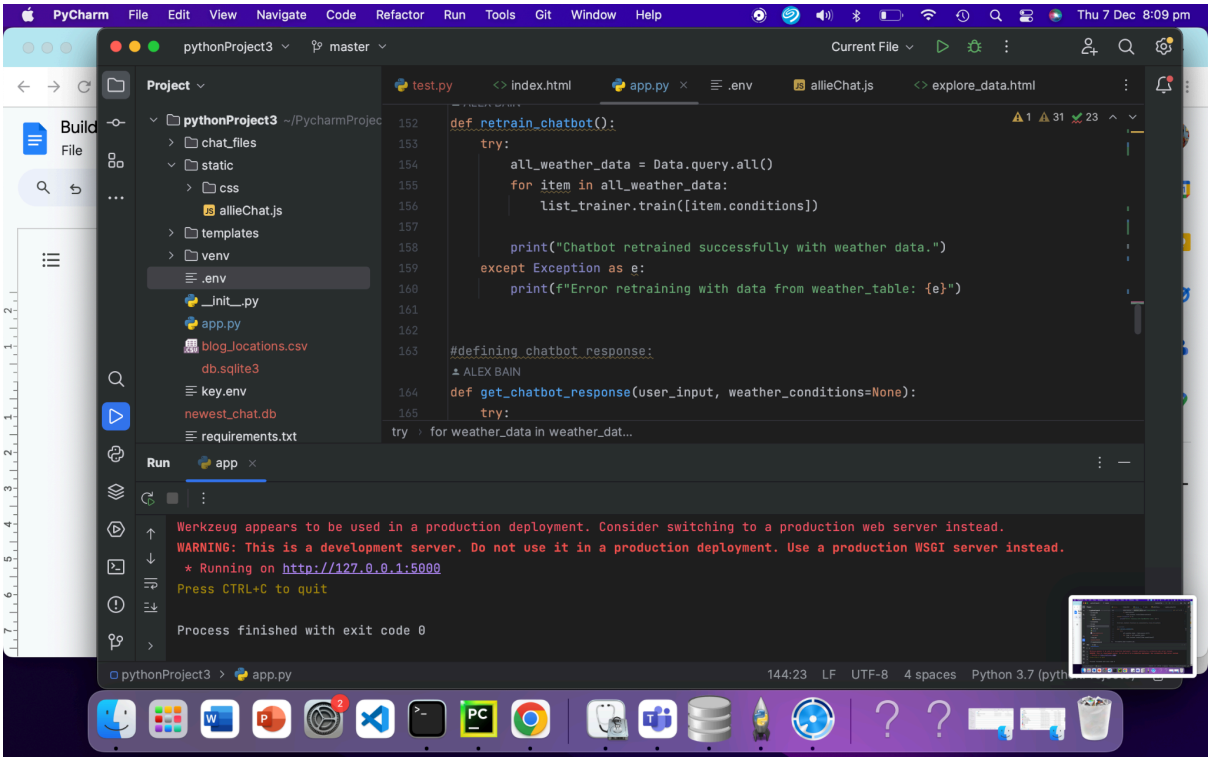
I wrote an index.html after doing some research on chatbot templates:



I wrote a basic chatbot route in app.py:

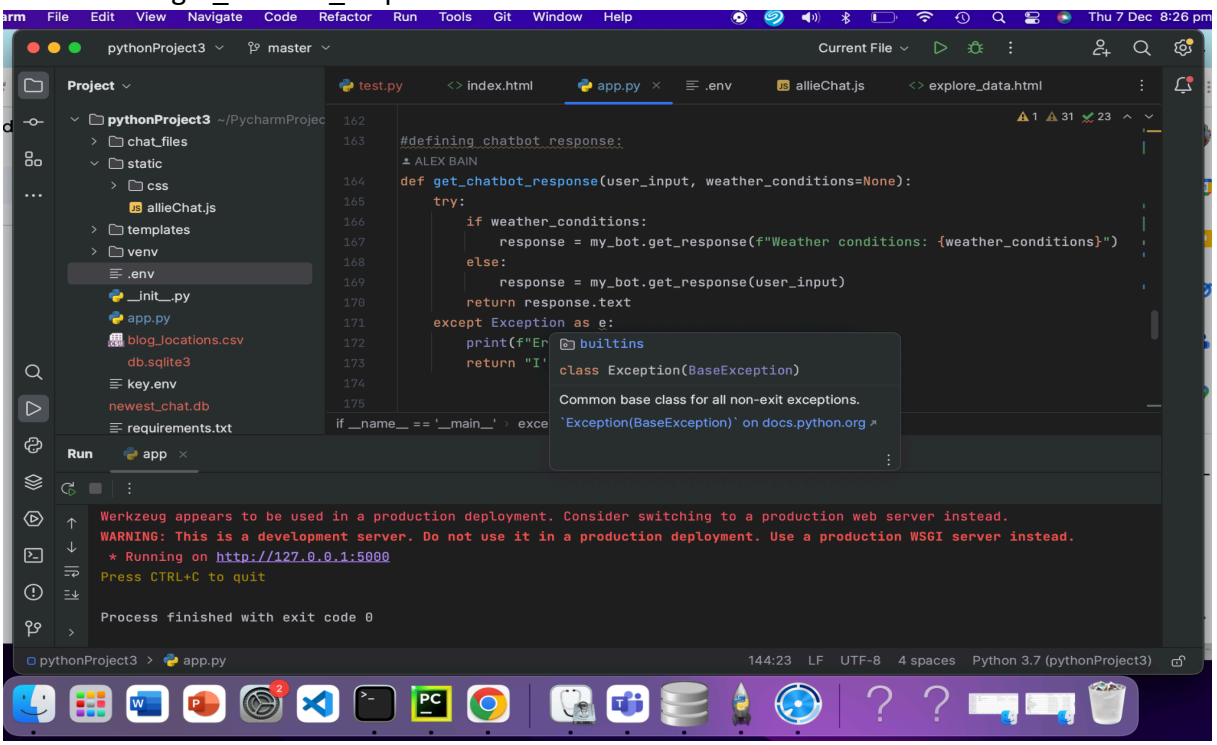


I implemented a retrain-chatbot route to train the bot as it collected user input and output responses:



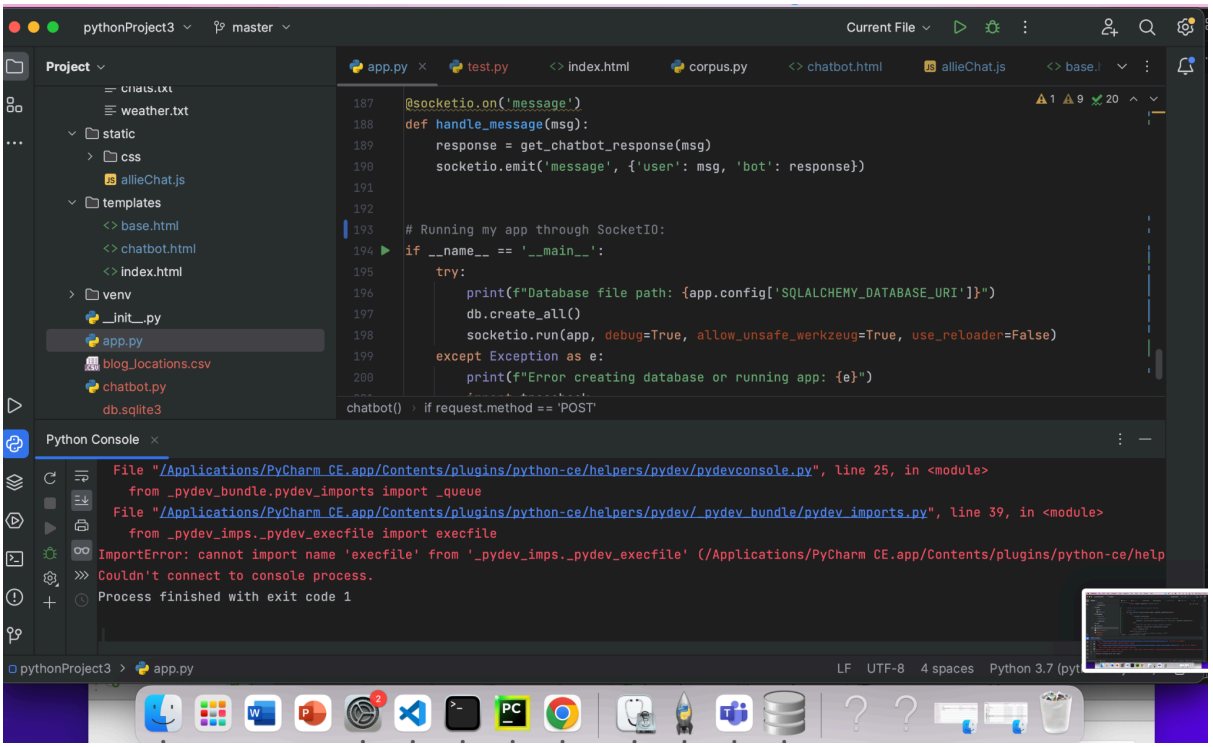
However, as the chatbot was slow, and training can still work without explicitly retraining it every run, I deleted this function, and instead focused on sufficiently training the bot without it.

I defined the get_chatbot_response:



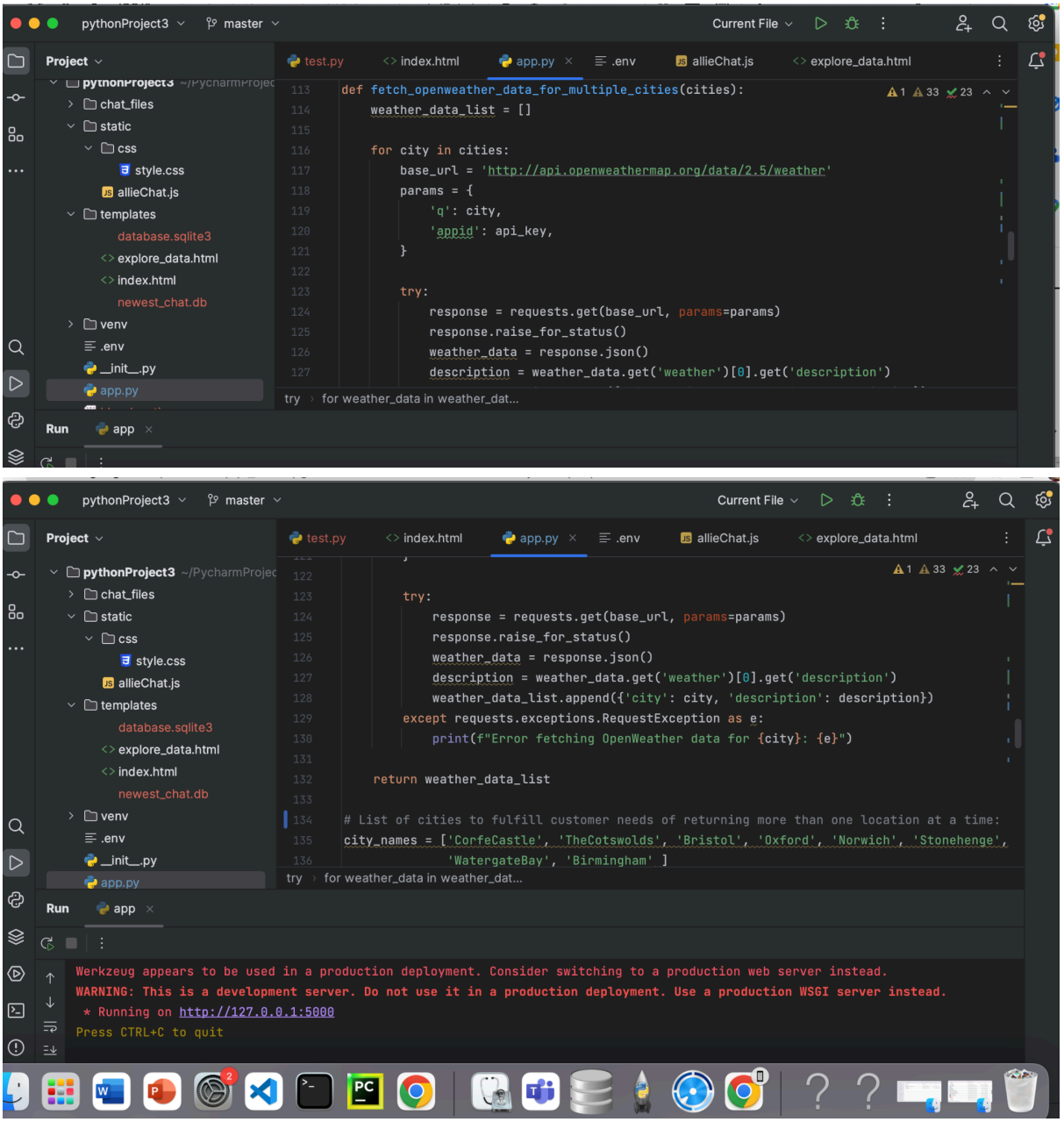
This helps to modularize my code and makes it easier to reuse the chatbot logic in different parts of the app.

Along with this and the app.route for chatbot, I created a Socket.io library route for the handling of the chatbot message, before running the app:

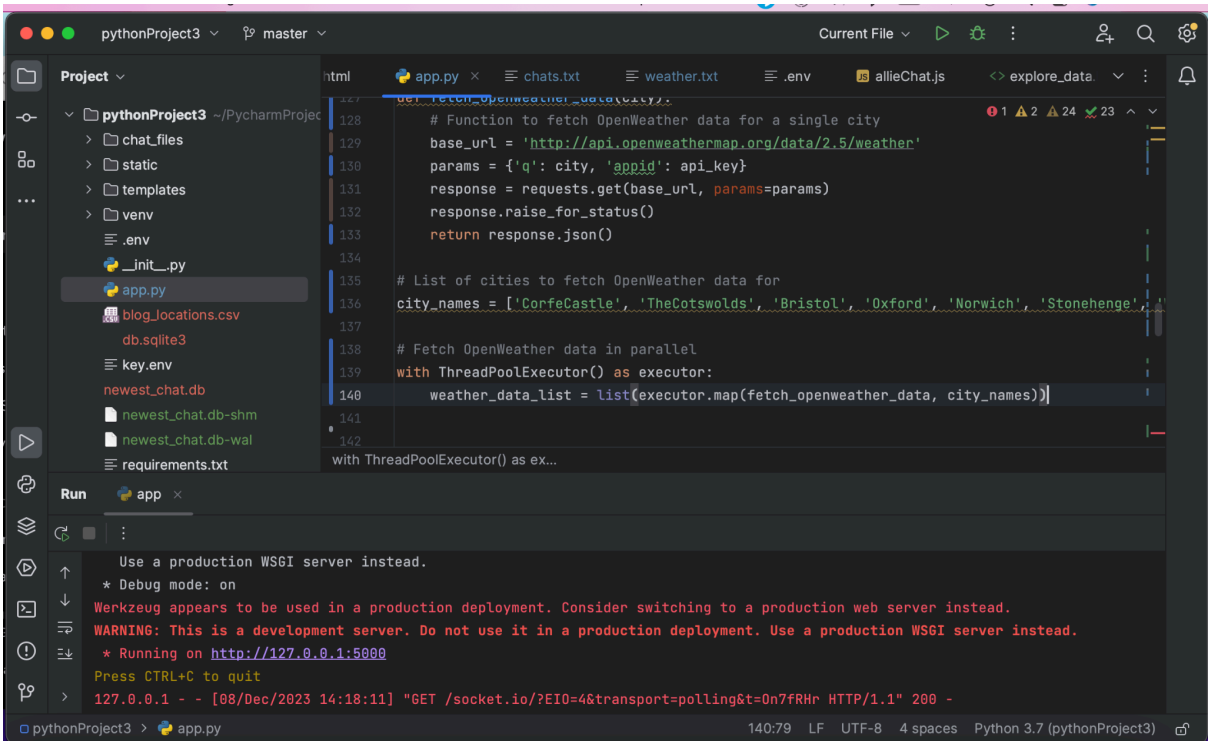


Pulling Data from OpenWeatherMaps API

As the current prototype can only return one location at a time, I have implemented an API through a function that calls to several locations at once and can collect information from multiple locations at the same time. I implemented a list for the function to loop over, along with my new API key:

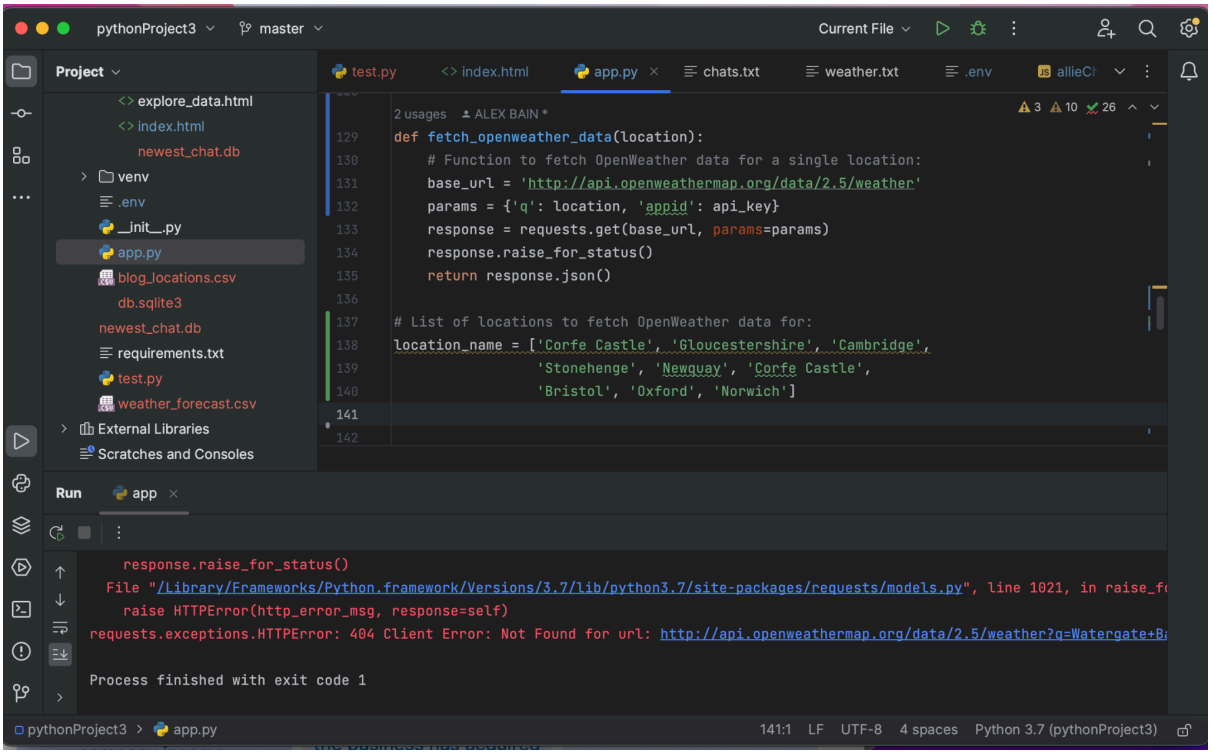


Despite deleting the retrain_bot function, AllieChat was still slow. I researched and imported ThreadPoolExecutor with parallel - and therefore more efficient- processing:



Thanks to the ThreadPoolExecutor() as executor the function now fetched OpenWeather data in parallel with the weather_data_list.

The Cotswolds and Watergate Bay returned errors, so instead I called from Gloucestershire and Newquay:



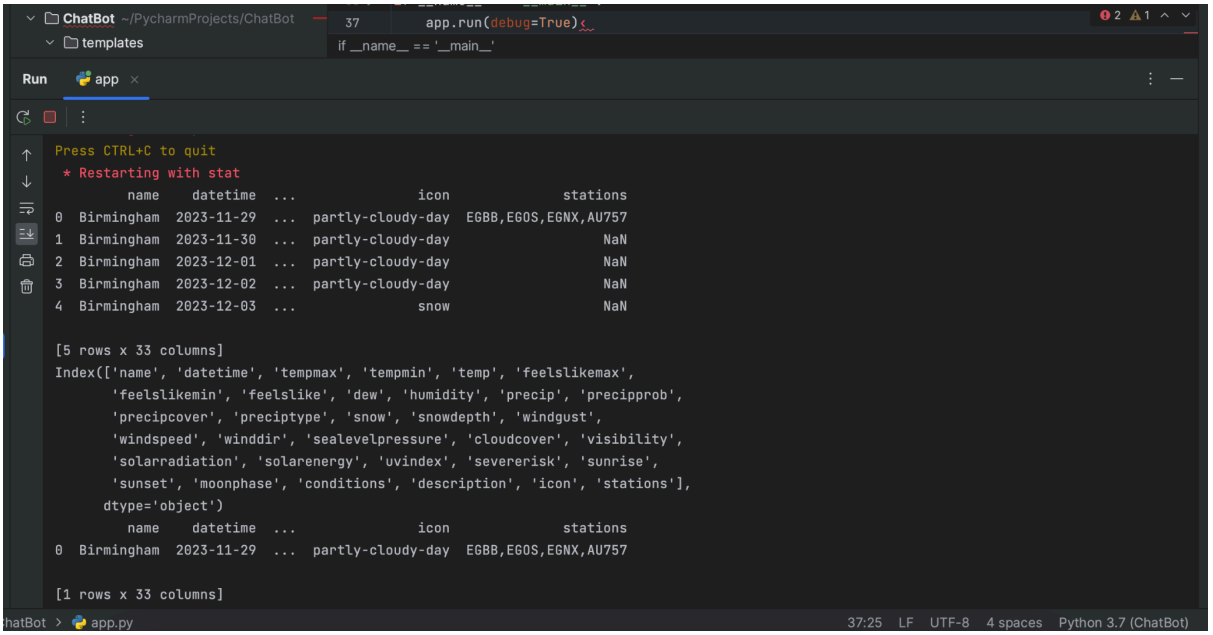
Pulling Data through CSV and Pandas

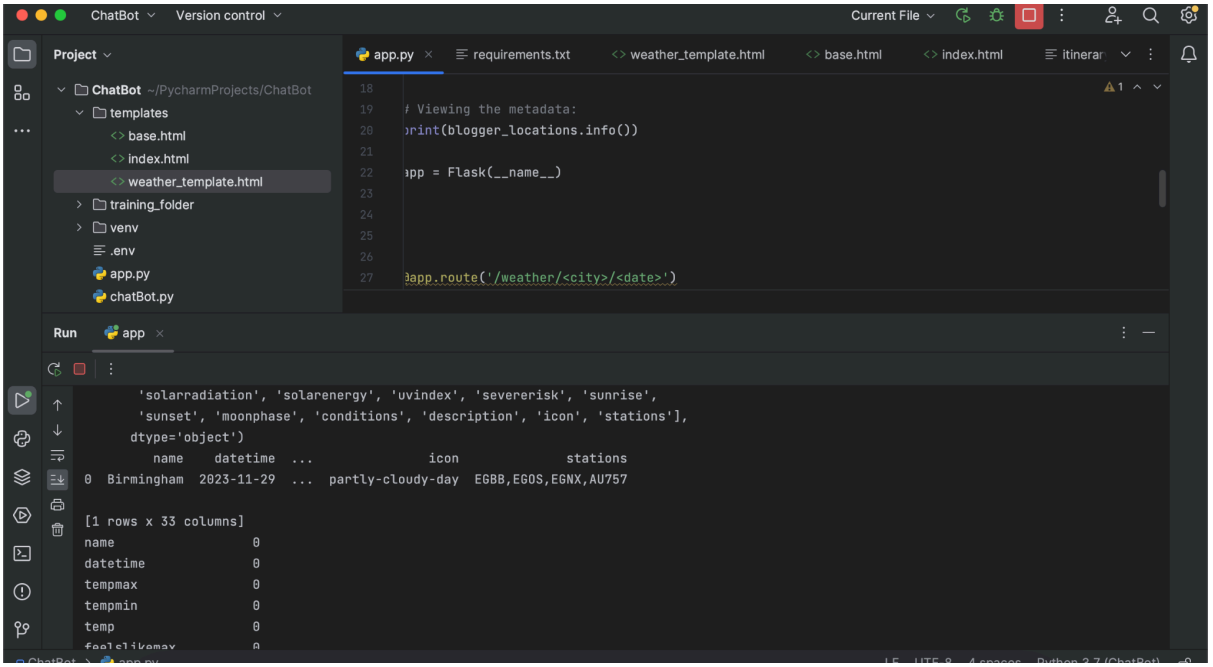
To pull data from the csv file, “weather_data”, using Pandas, I first worked on data retrieval and reading using Pandas to test my data.

First, I created a csv file (see **Wrangling Data Using DB Browser and SQLite,** page 18**)**. Using pd.read_csv to import itinerary.csv, and head() and columns() to display, I retrieved data using python Object-Oriented Programming methods in a separate py file for testing:

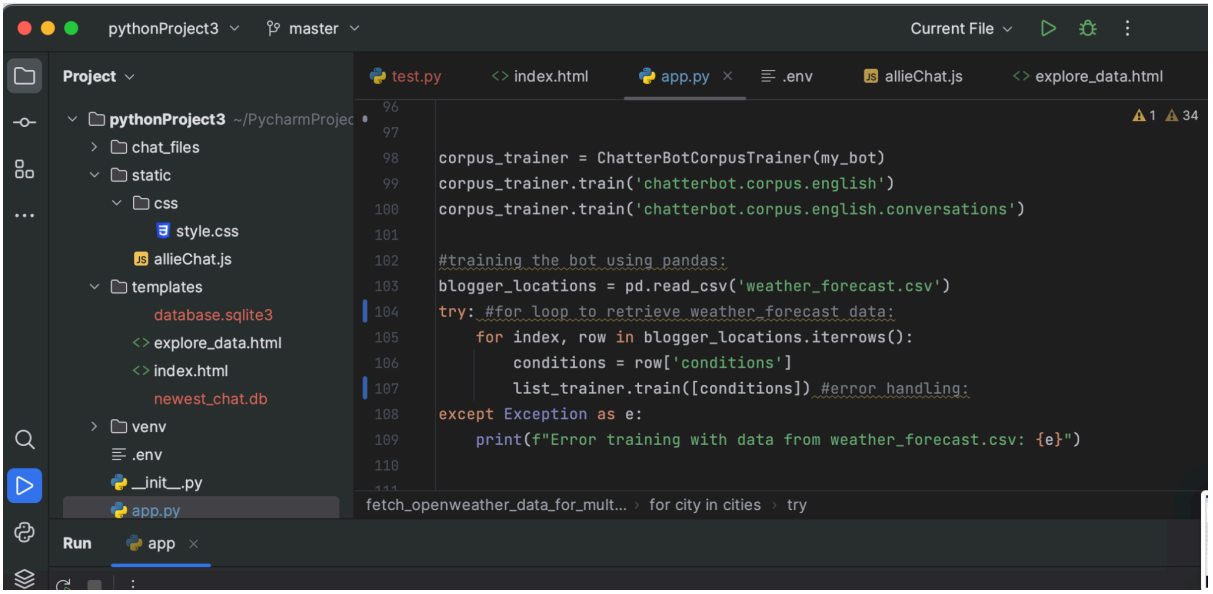
```
#Importing the data sets:
blogger_locations = pd.read_csv('blog_locations.csv')
print(blogger_locations.head())
print(blogger_locations.columns)

birmingham_weather = blogger_locations[(blogger_locations['name'] == 'Birmingham') &
(blogger_locations['datetime'] == '2023-11-29')]

# Viewing the output:
print(birmingham_weather)

# Exploring the data.
# Any missing values?
print(blogger_locations.isna().sum())


# Viewing the metadata:
print(blogger_locations.info())


app = Flask(__name__)

@app.route('/weather/<city>/<date>')
def display_weather(city, date):
  # Filter weather data based on the requested city and date
  city_weather = blogger_locations[(blogger_locations['name'] == city) &
(blogger_locations['datetime'] == date)]

  if not city_weather.empty:
    # Pass relevant information to the template
    return render_template('weather_template.html', city=city, date=date,
temperature=city_weather.iloc[0]['temp'], conditions=city_weather.iloc[0]['conditions'])
  else:
    return render_template('weather_not_found.html', city=city, date=date)

if __name__ == '__main__':
```

app.run(debug=True)



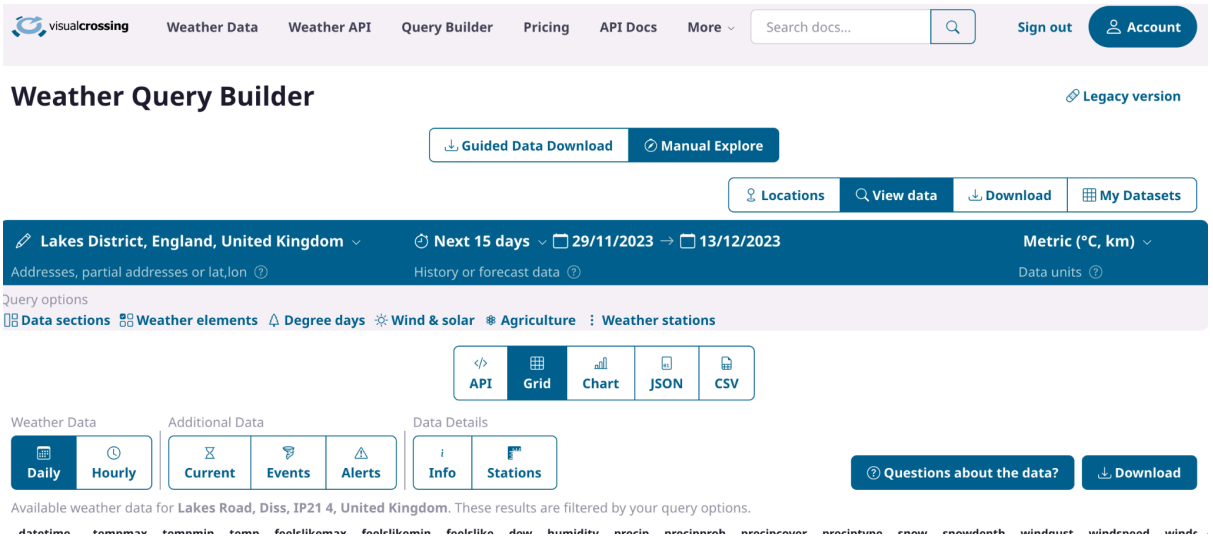


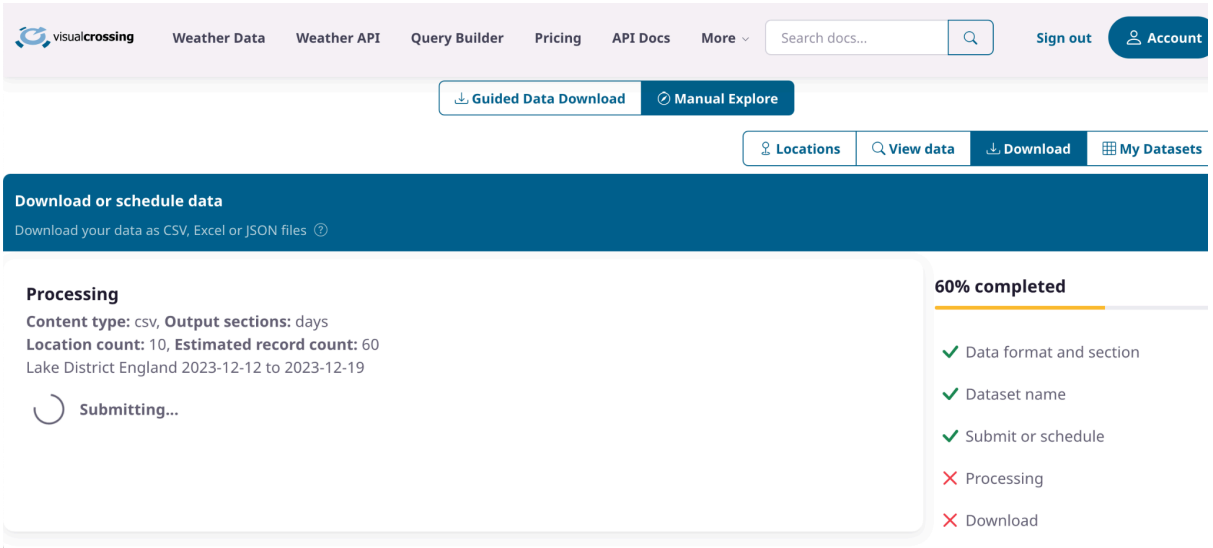
I implemented the dataframes into the app by calling from blogger_locations =pd.read_csv('weather_forecast.csv') at the top of the file, and a try-except error block:



Wrangling Data Using DB Browser and SQLite

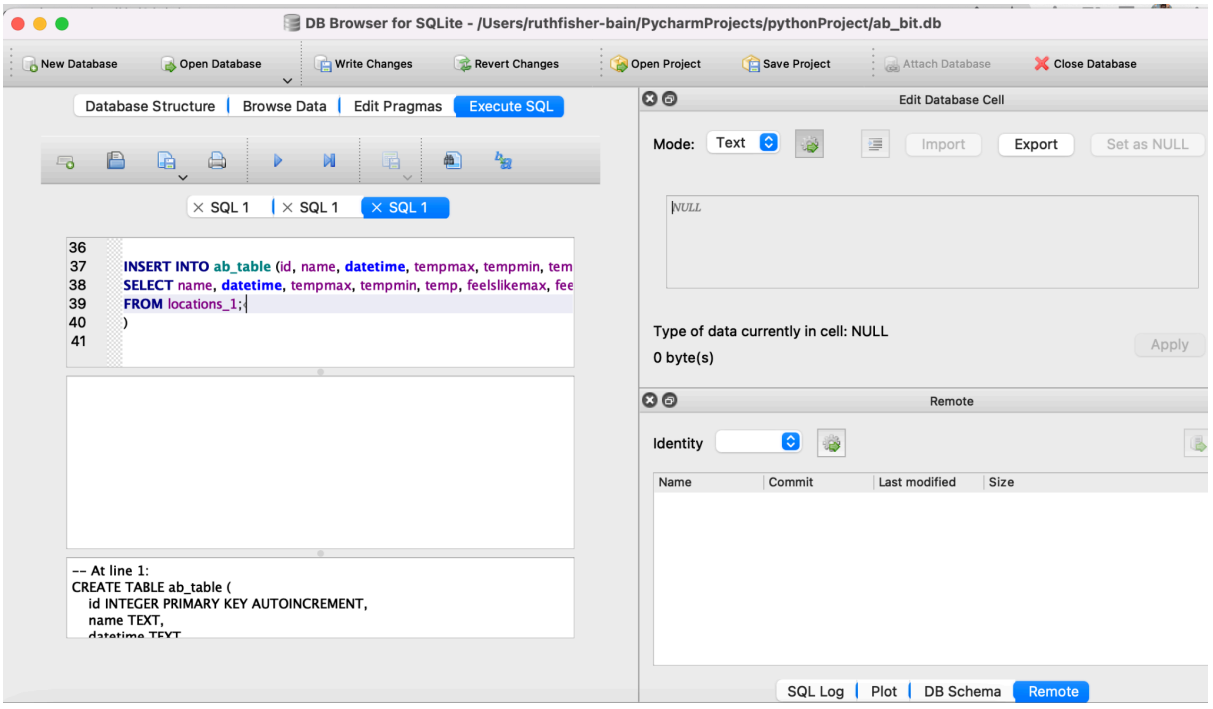
To solve the company's issue of only returning daily weather data, I used Visual Crossing global weather database's Weather Query Builder. I created a csv file by inputting the locations on the blogger's itinerary over a two-week period from a future date (note: the location Lake District here, but this wasn't specifically called in the app):





DB Browser for SQLite helped me identify how the different components of the app and database worked together, and to more clearly decipher any issues I had or errors I received.

Once I downloaded the csv, a relational database was easy to logically design from my initial concept through DBMS SQLite, ensuring accuracy, consistency and durability. Physically, this was more challenging. I added the id heading into the table, as this wasn't in the original csv code, via commands, incorporated this into the app, and investigated errors carefully. Data was displayed from both weather_table and locations_data:



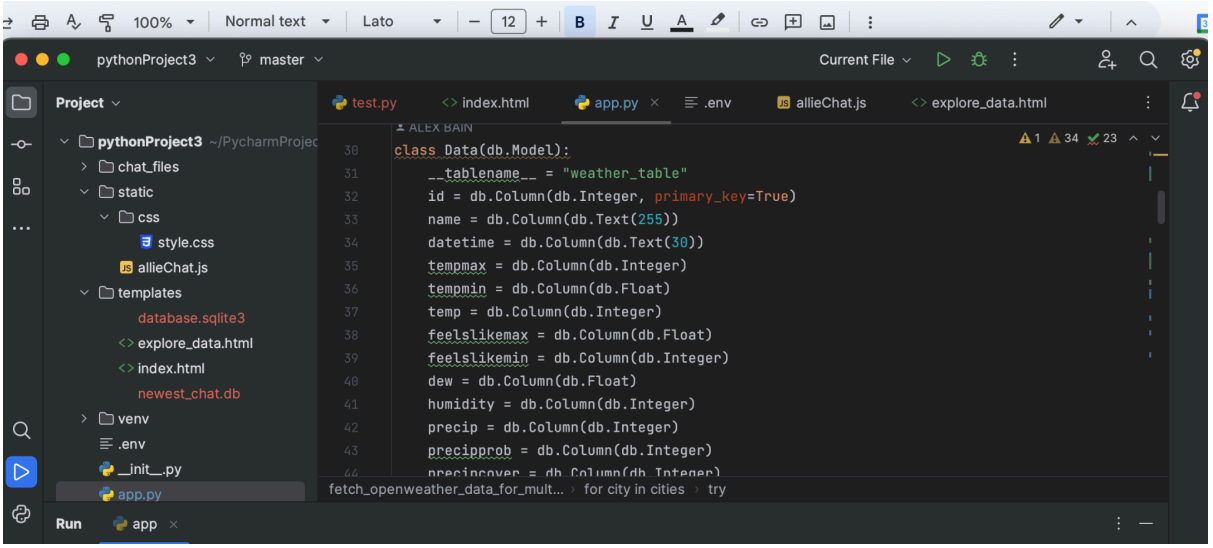
I added in an exception to ensure that I was retrieving data correctly, which helped heaps when I realised I had not completed the Write of the newest_chat.db database:

```
Exception as e:
  print(f"Error creating database: {e}")
  # Add the following line to print the stack trace
  import traceback
  traceback.print_exc()
```

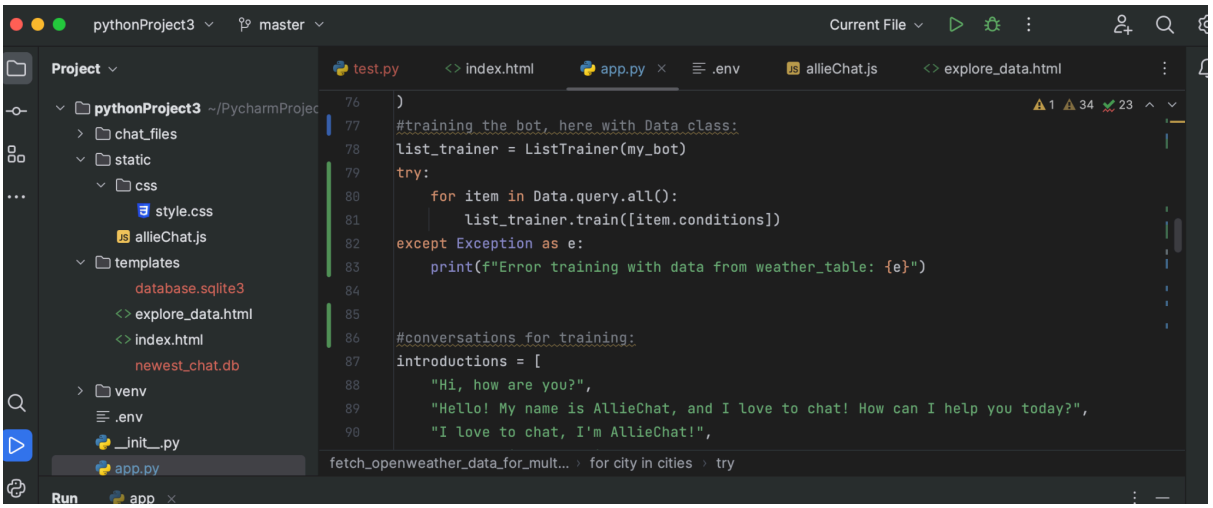
Eventually, I was able to replicate the database in my app via an “index.html” template, and this told me that the app and browser were reading the data correctly.

Calling Data from Data Class:

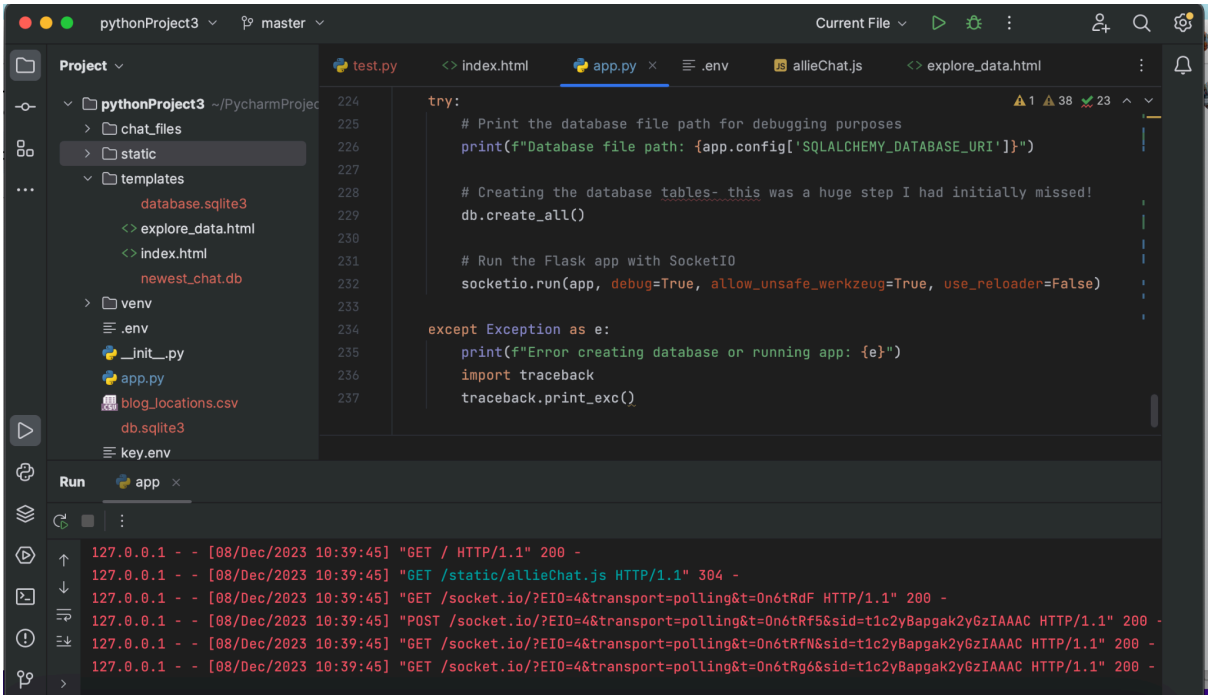
Now that my database had been properly created, I created a Data class in my app.py file:



I called trained the bot using ListTrainer and data from Data class:

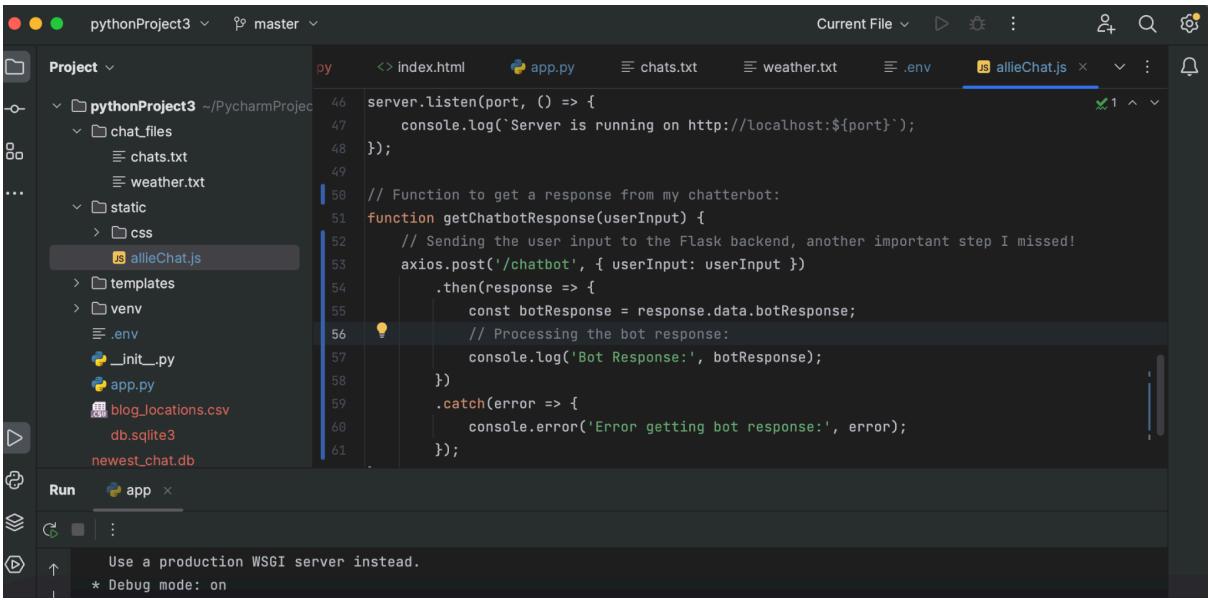


I added in db.create_all() to create database tables as good code practice is placed above the app's run statement,which includes debugging and calls from Socket.IO:

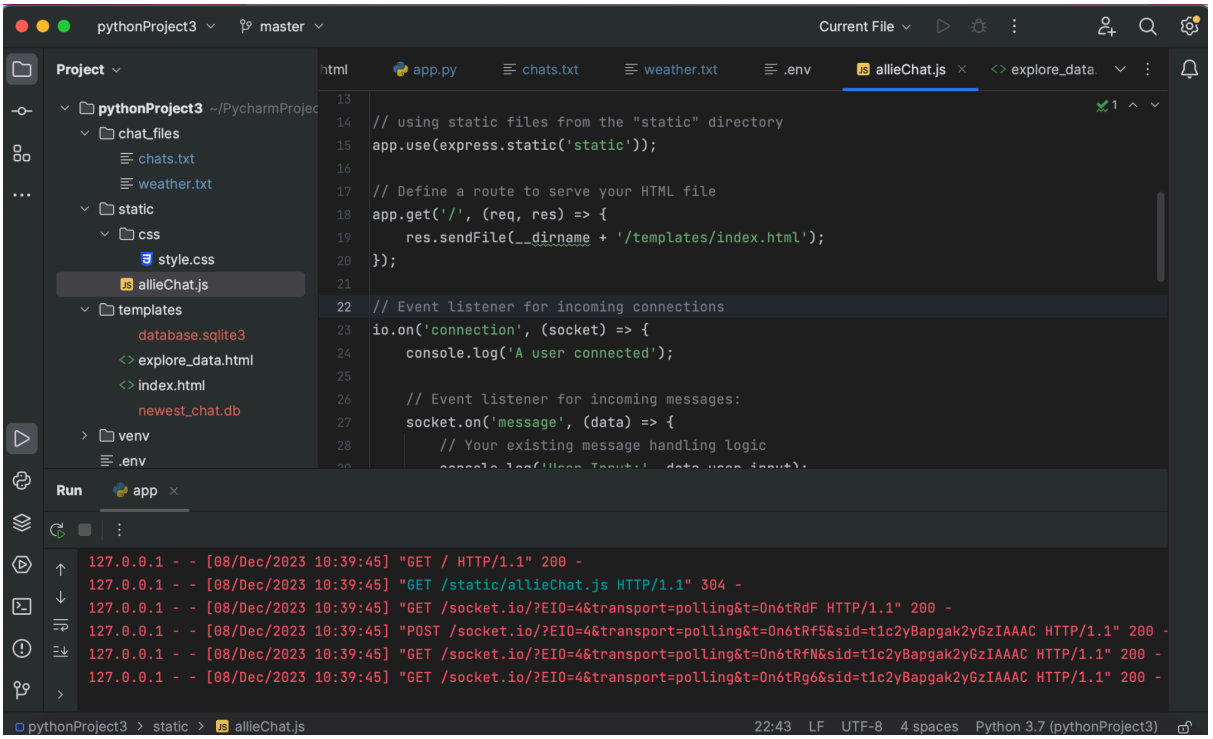


Chatbot Javascript

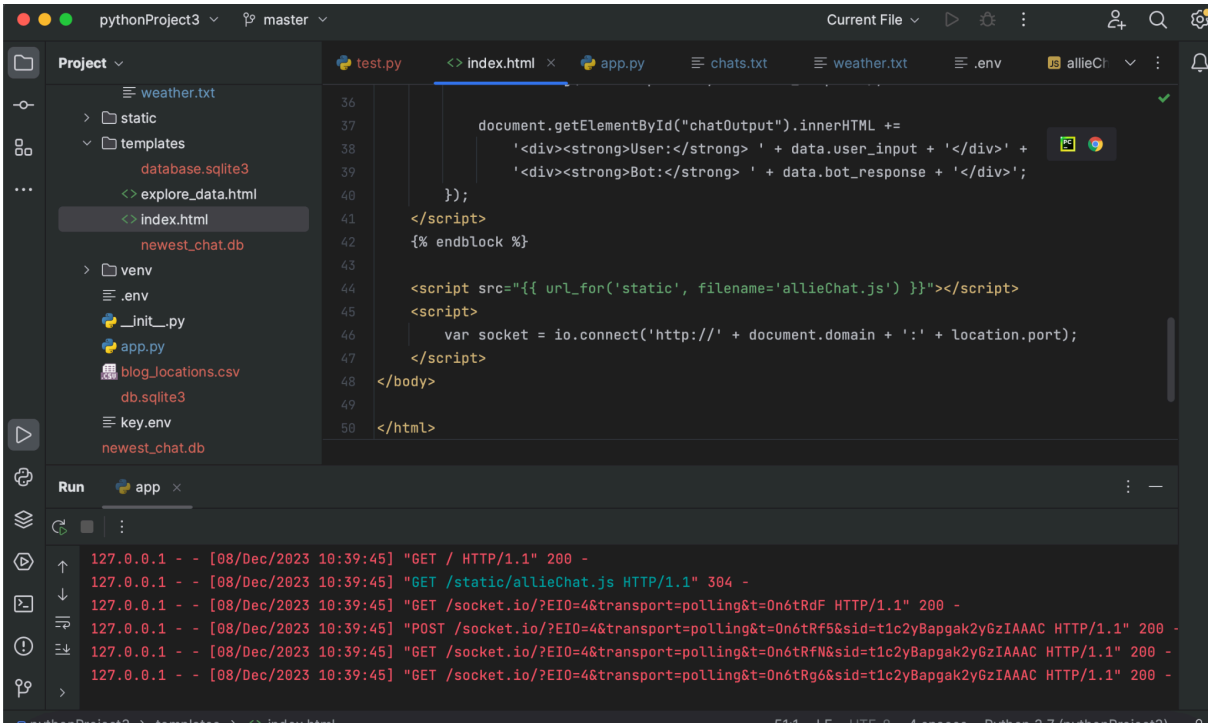
My initial simple JS code was slow or unresponsive. After a fair bit of testing in a test.py file, I sought some advice from Gita, and realised the event listener I'd called from js wasn't receiving the request. My script wasn't sending the user input to the Flask backend. I had forgotten to replace the placeholder string with a HTTP POST request to a server endpoint, which I amended using the Axios library:

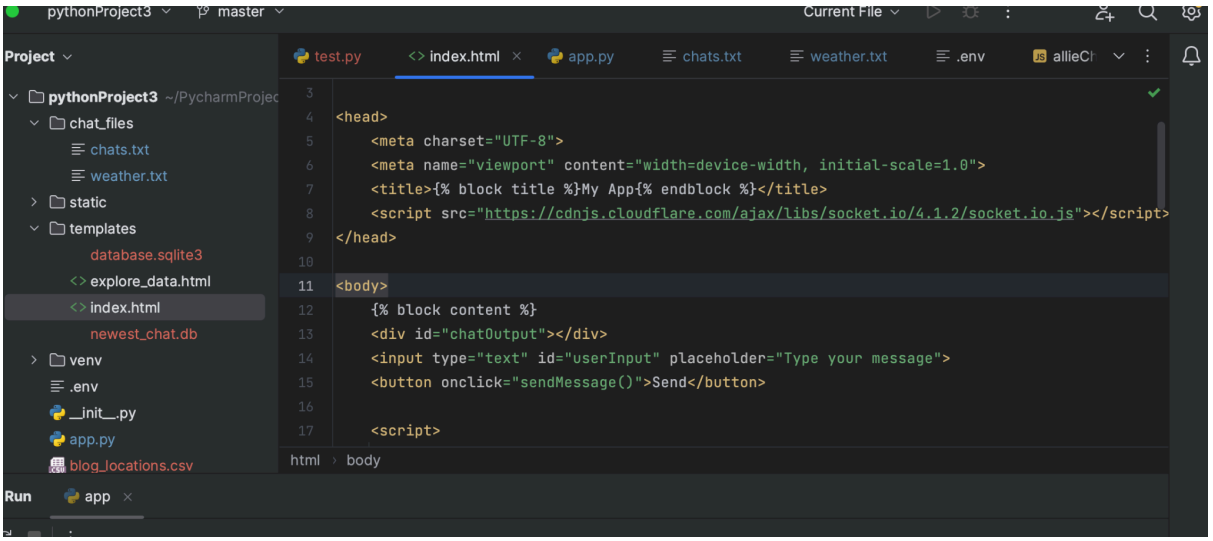


I tested chatbot output in a separate project, and uninstalled then reinstalled Socket.IO . I added in the static path and socket.io requirement, which were initially missing in my js:

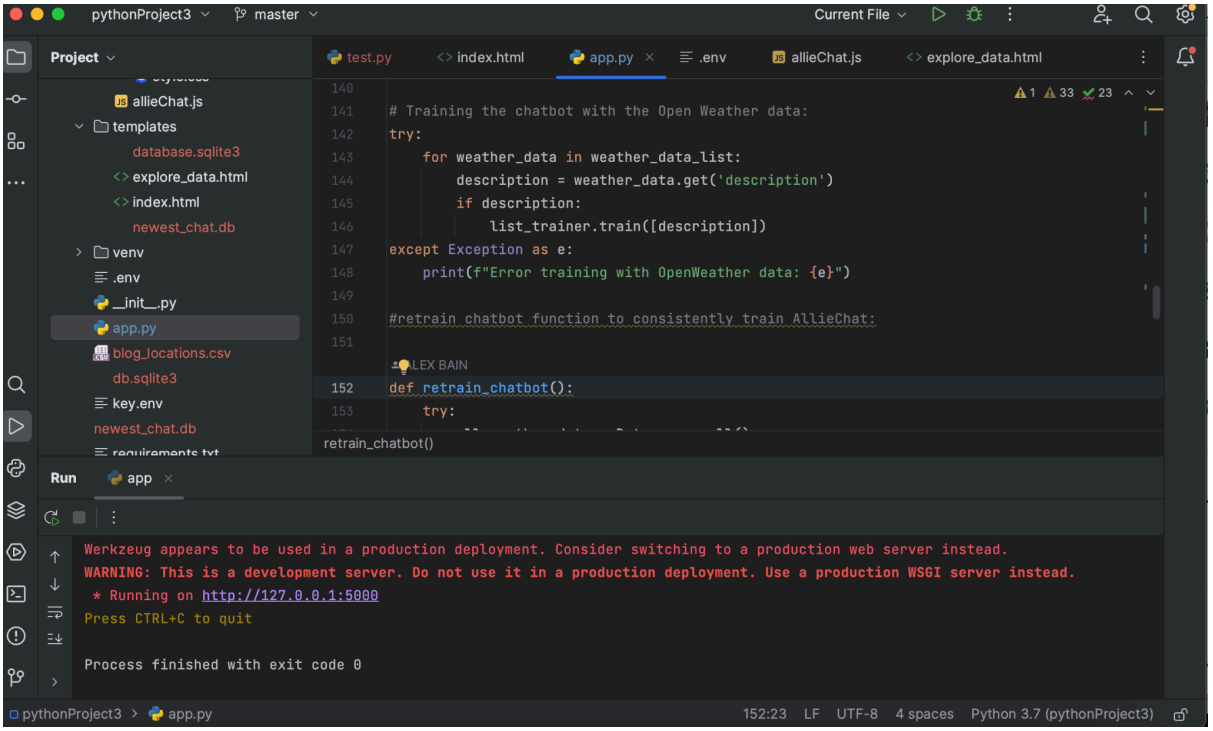


My output showed I had successfully established WebSocket connections via Socket.IO. I ensured my index.html called the chatbot properly, including JS and Socket.IO links:

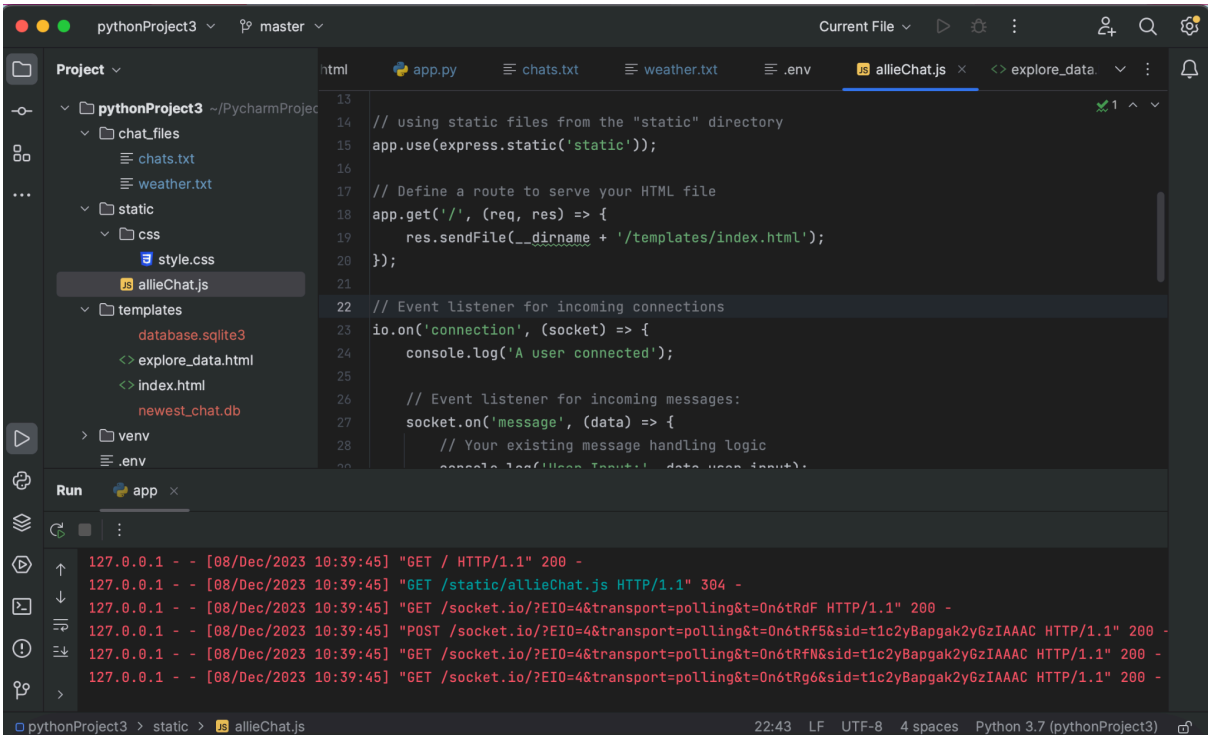




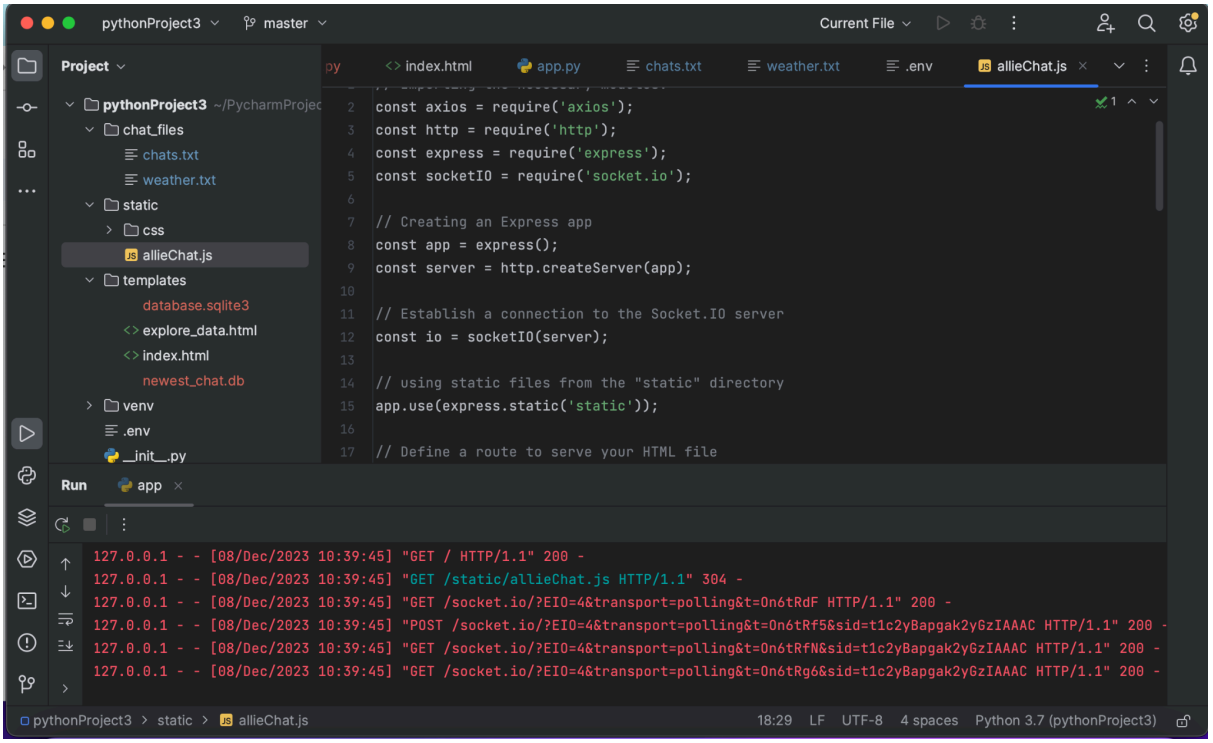
I added some more error handling to my functions including the Exception method to check I was pulling the correct data:

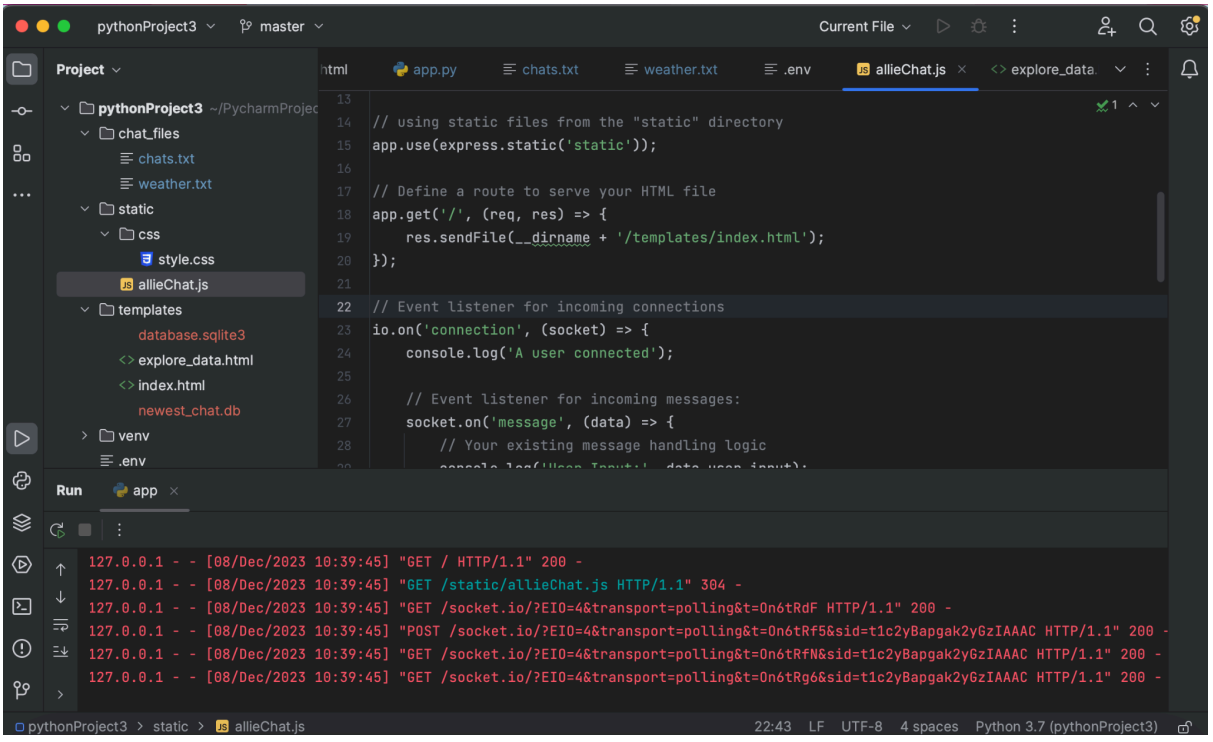


I realised I had identified the incorrect file path for the app.get Javascript method and changed this to link to templates/index.html:

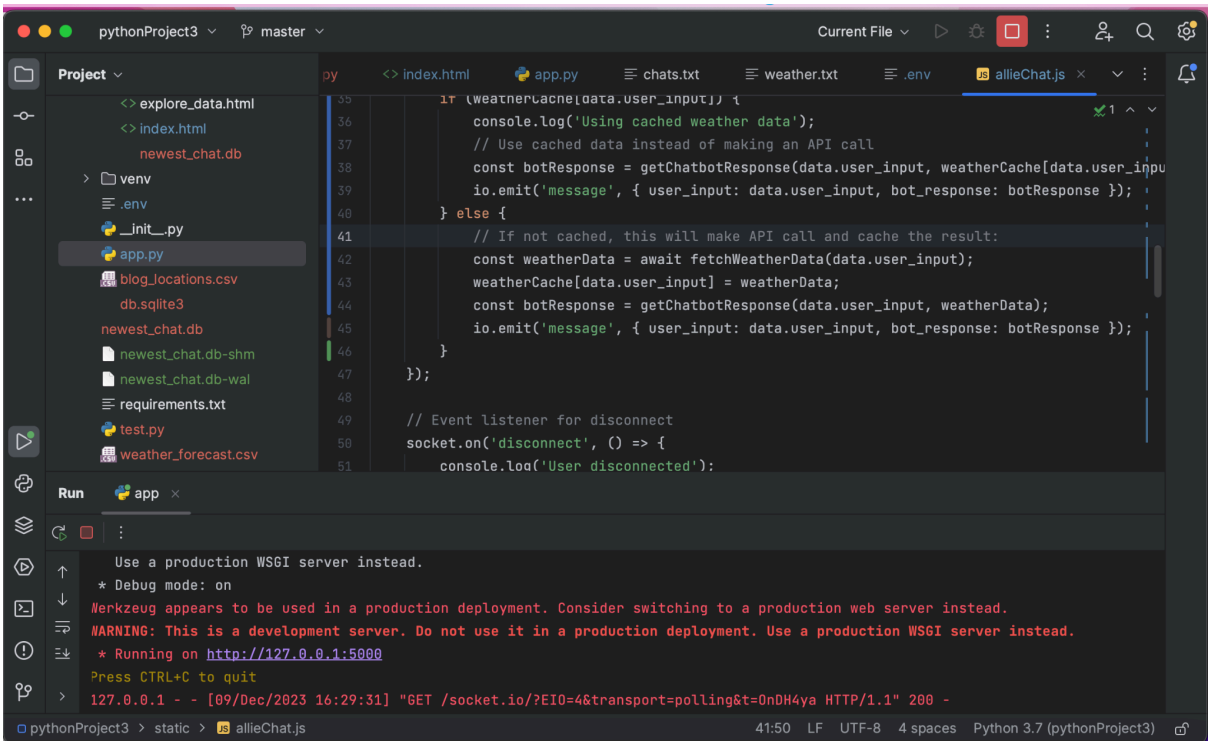


This was good practice in using JS again, as I needed to edit my file quite a few times before I saw any result, and make sure the Socket.IO and allieChat.js link were both placed in the index.html file correctly:

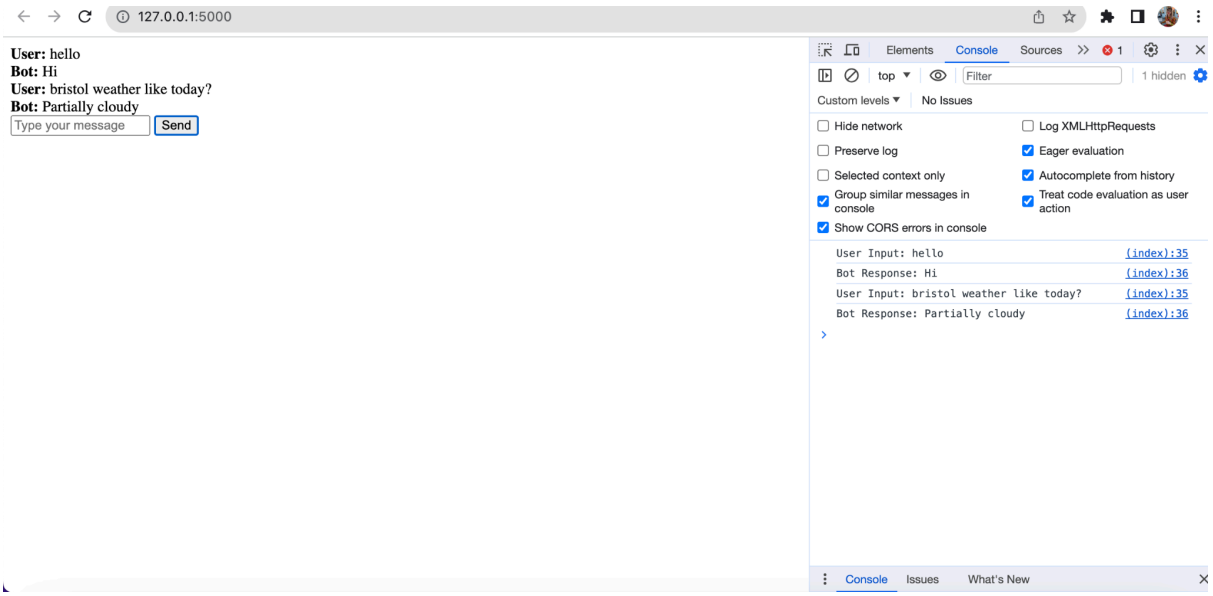


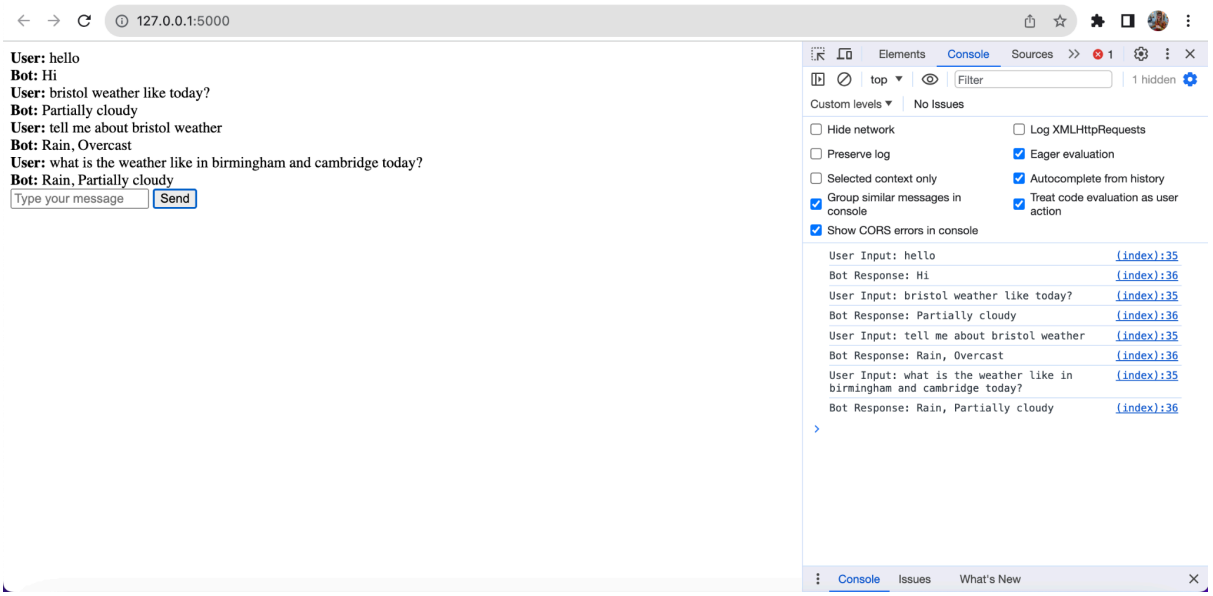


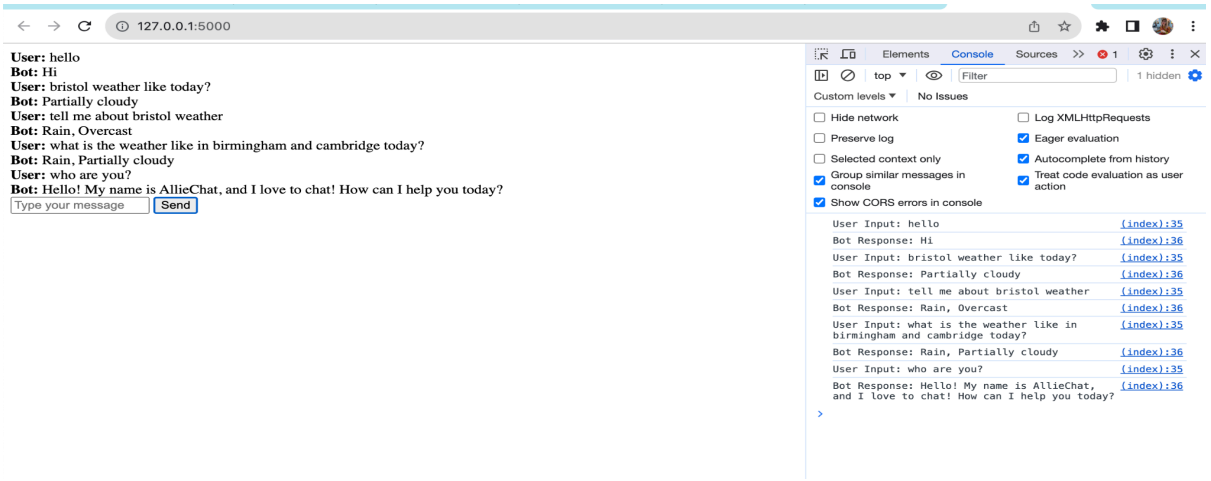
To rectify the company not budgeting for additional costs from OpenWeather API if site traffic increases, I created a ‘weatherCache’ object. When a user sends a message, the server checks if the weather data is already cached, and if it is, server uses the cached data; otherwise, it makes an API call, caches the result, and then uses it:



I ran the bot and found my chatbot now pulls from my data locations in a chat format:




127.0.0.1:5000
User: hello
Bot: Hi
User: bristol weather like today?
Bot: Partially cloudy
User: tell me about bristol weather
Bot: Rain, Overcast
User: what is the weather like in birmingham and cambridge today?
Bot: Rain, Partially cloudy
Type your message
Send
Elements
Console
Sources
Filter
Custom levels
No Issues
Hide network
Log XMLHttpRequests
Preserve log
Eager evaluation
Selected context only
Autocomplete from history
Group similar messages in console
Treat code evaluation as user action
Show CORS errors in console
User Input: hello
Bot Response: Hi
User Input: bristol weather like today?
Bot Response: Partially cloudy
User Input: tell me about bristol weather
Bot Response: Rain, Overcast
User Input: what is the weather like in birmingham and cambridge today?
Bot Response: Rain, Partially cloudy
Console
Issues
What's New


127.0.0.1:5000
User: hello
Bot: Hi
User: bristol weather like today?
Bot: Partially cloudy
User: tell me about bristol weather
Bot: Rain, Overcast
User: what is the weather like in birmingham and cambridge today?
Bot: Rain, Partially cloudy
User: who are you?
Bot: Hello! My name is AllieChat, and I love to chat! How can I help you today?
Type your message
Send
Elements
Console
Sources
Filter
Custom levels
No Issues
Hide network
Log XMLHttpRequests
Preserve log
Eager evaluation
Selected context only
Autocomplete from history
Group similar messages in console
Treat code evaluation as user action
Show CORS errors in console
User Input: hello
Bot Response: Hi
User Input: bristol weather like today?
Bot Response: Partially cloudy
User Input: tell me about bristol weather
Bot Response: Rain, Overcast
User Input: what is the weather like in birmingham and cambridge today?
Bot Response: Rain, Partially cloudy
User Input: who are you?
Bot Response: Hello! My name is AllieChat, and I love to chat! How can I help you today?

3. Reflection

I found the building of my chatbot to be a challenging yet rewarding experience.

When it comes to parallel programming, there is so much going on to keep track of, and I found that, as per my learning, a separation of concerns was really important to the design process.

I learned that research, trying new methods and running the app over and over again- with a clear, singular focus per component- helps in becoming a better programmer. This helped me to relax into testing and running into errors as a part of the process. Under less pressure, I was able to code better. In further iterations, I would like to speed up the app as well as incorporate it into my initial weather application.