# Implementation:

## Database Implementation and Connection:

The database was set up using SQLalchemy which is a library inside of python. This enabled me to make the dataset dynamically by just making the models of each entity.

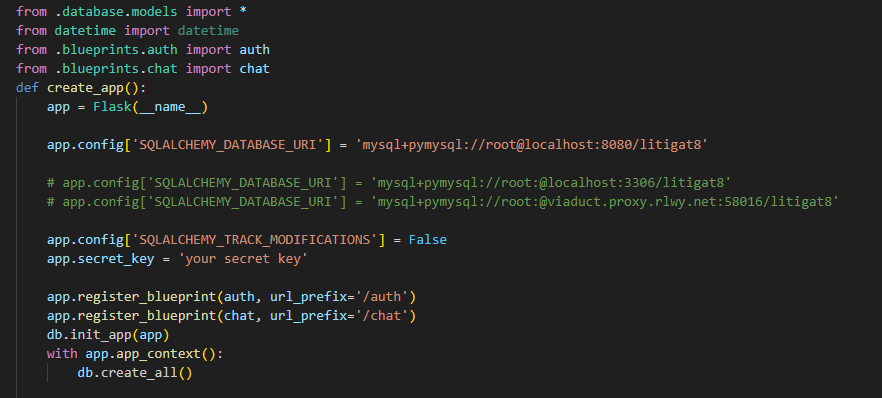


Figure 28 Showing the Setup of Database Config

With the database initiation its assigned to the flask application by setting up the correct URL, username and password for the database inside of the SQL database. The variable **db** can be used to access any of the model i.e User, ChatSession and Conversation. Making the overall access of the database really easy instead of using SQL statements running directly on MySQL.

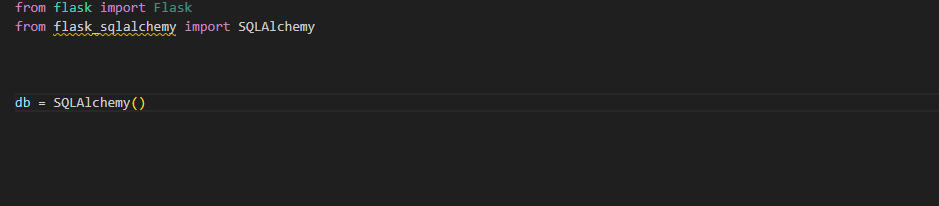


Figure 29 Setting up the Database

The models set up are made on accordance with the database schema which is displayed in the Figure 25. All of them have a primary key but ChatSession and Conversation have foreign keys and one-to-many relationship with the User\_ID.



Figure 30 Setting up the Models for Database

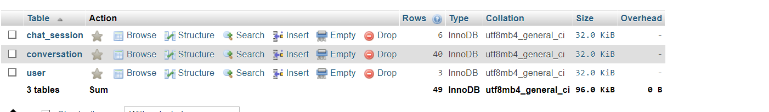


Figure 31 Showing the constructed Table in the Litigate Database

The chapter concludes the setting up of the database for Litigat8 application.

## Front-end Implementation:

**Main Home Page:**

The main page is designed based on the initial approach of keeping the user interface as intuitive as possible while keeping the functionality of the page. The initial design was designed using wireframes to make sure the it has all the essential components are present in the page.

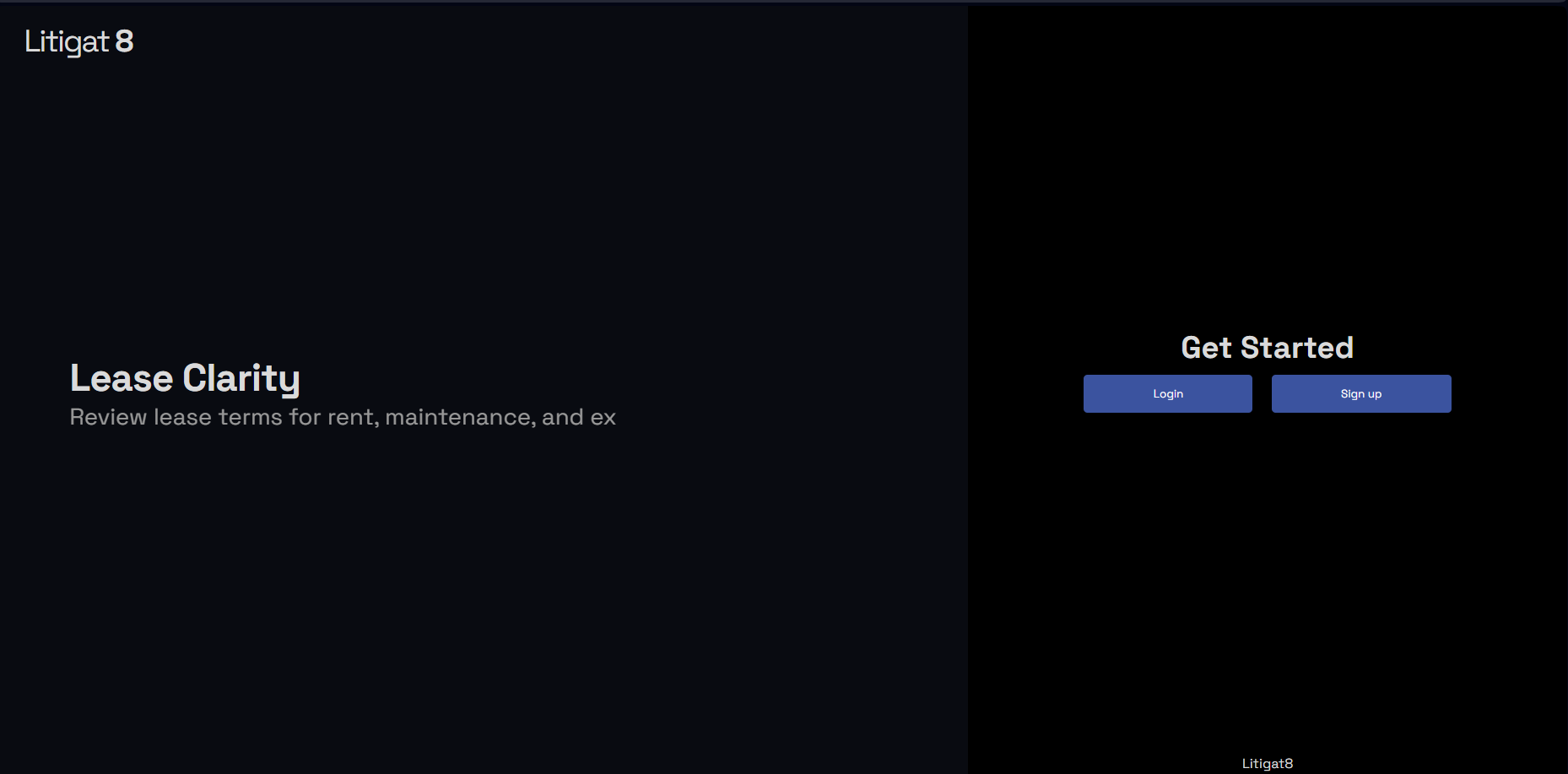
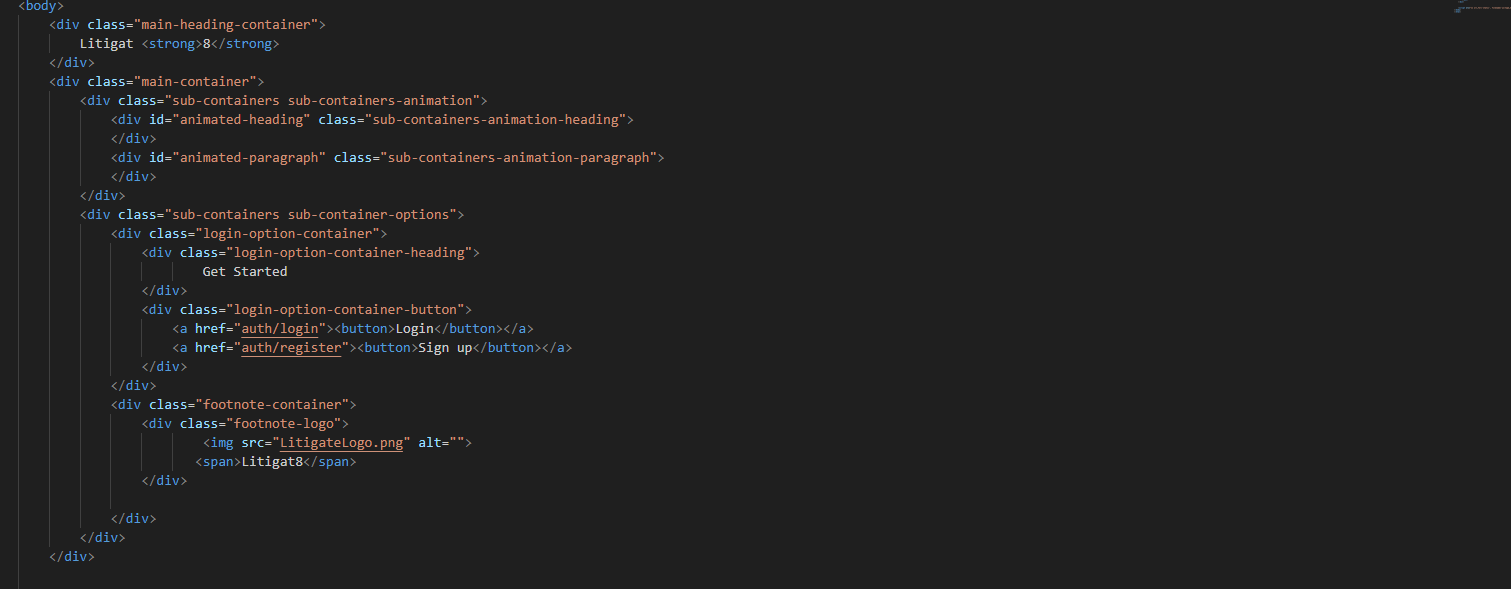


Figure 32 Showing the Front-end Implementation of Main-Page

An additional feature of typewriting was implemented as well as well to give it a bit more dynamic look.



**Login And Sign-Up Page:**

Following on the intuitive and approachable design of the pages, the design was based on the wireframes that was designed earlier. With two input field in login page and three input field in resistration page asking for the desired input from the users.

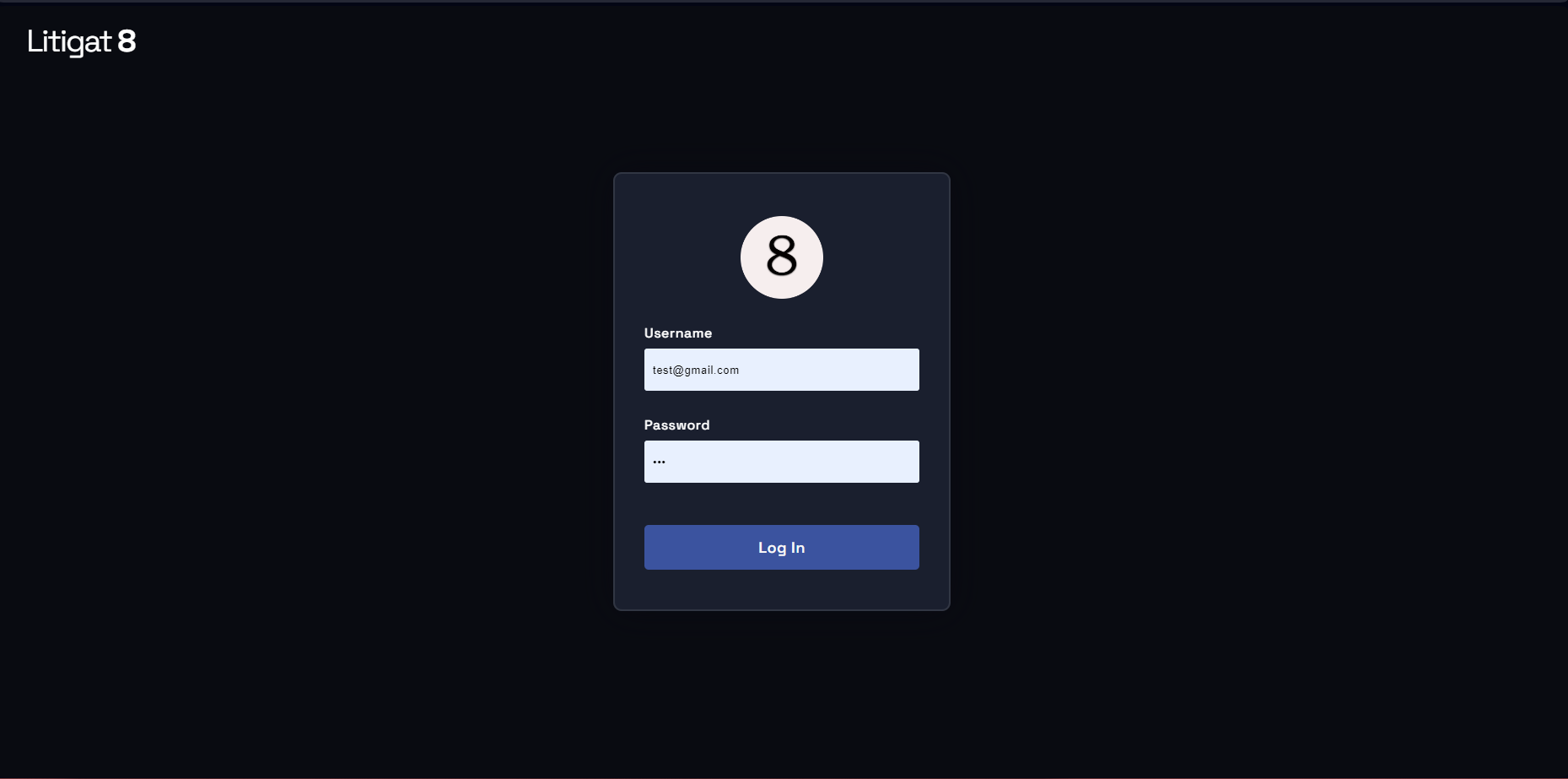


Figure 33 Showing the Front-end Implementation of Login Page of Litigate



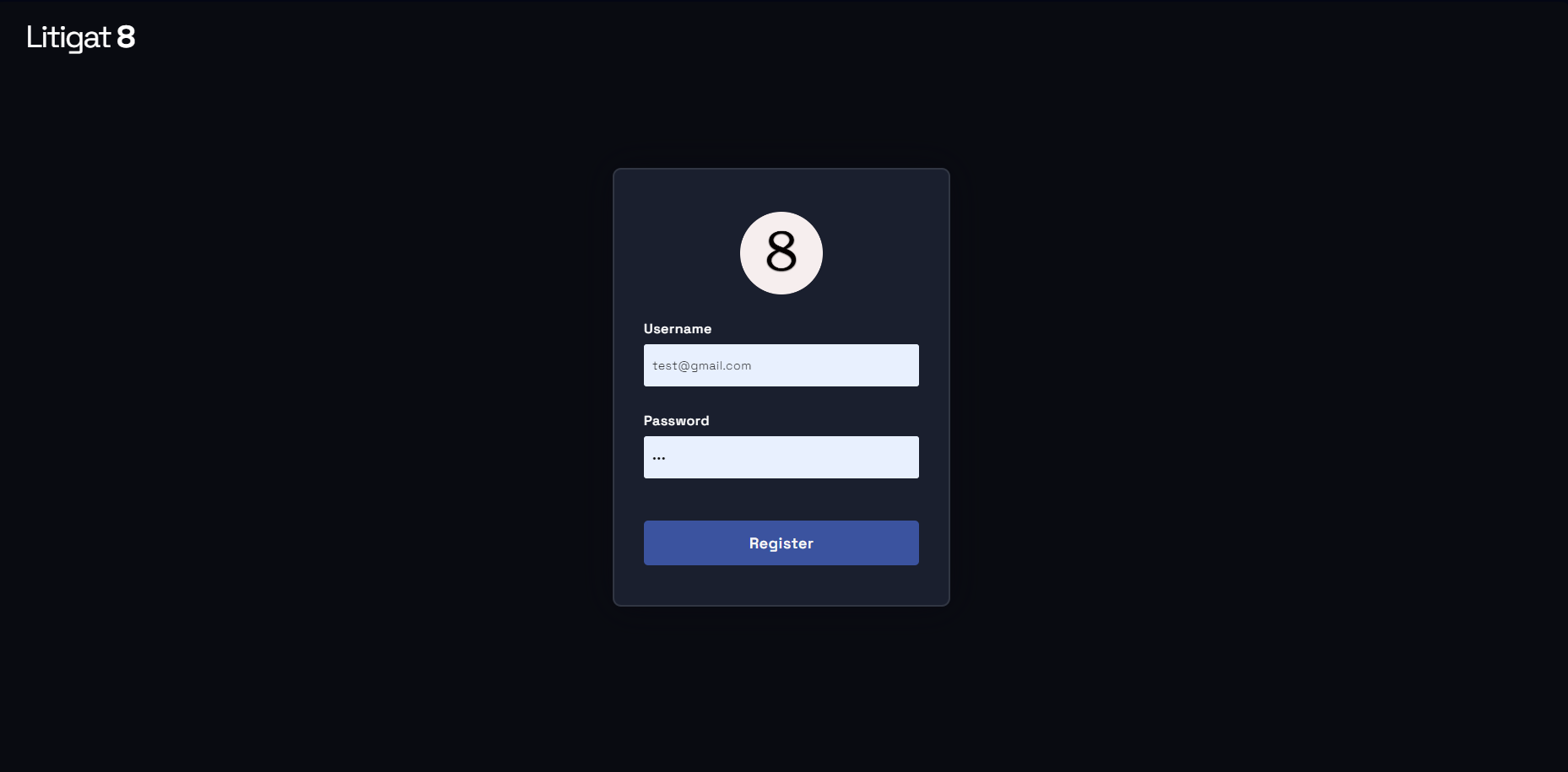


Figure 34 Showing the Front-end Implementation of Register Page of Litigate



**Chat Interface:**

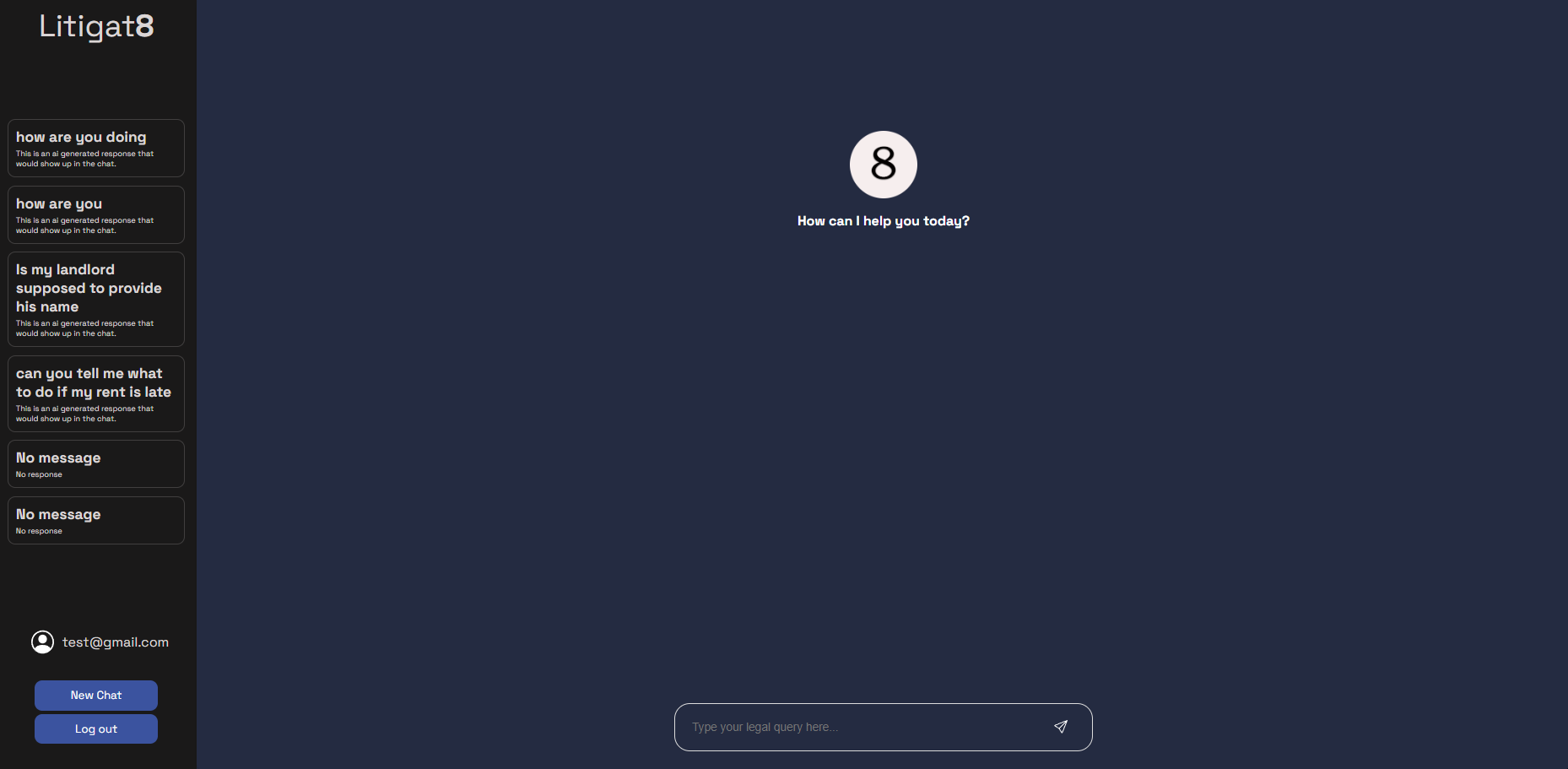
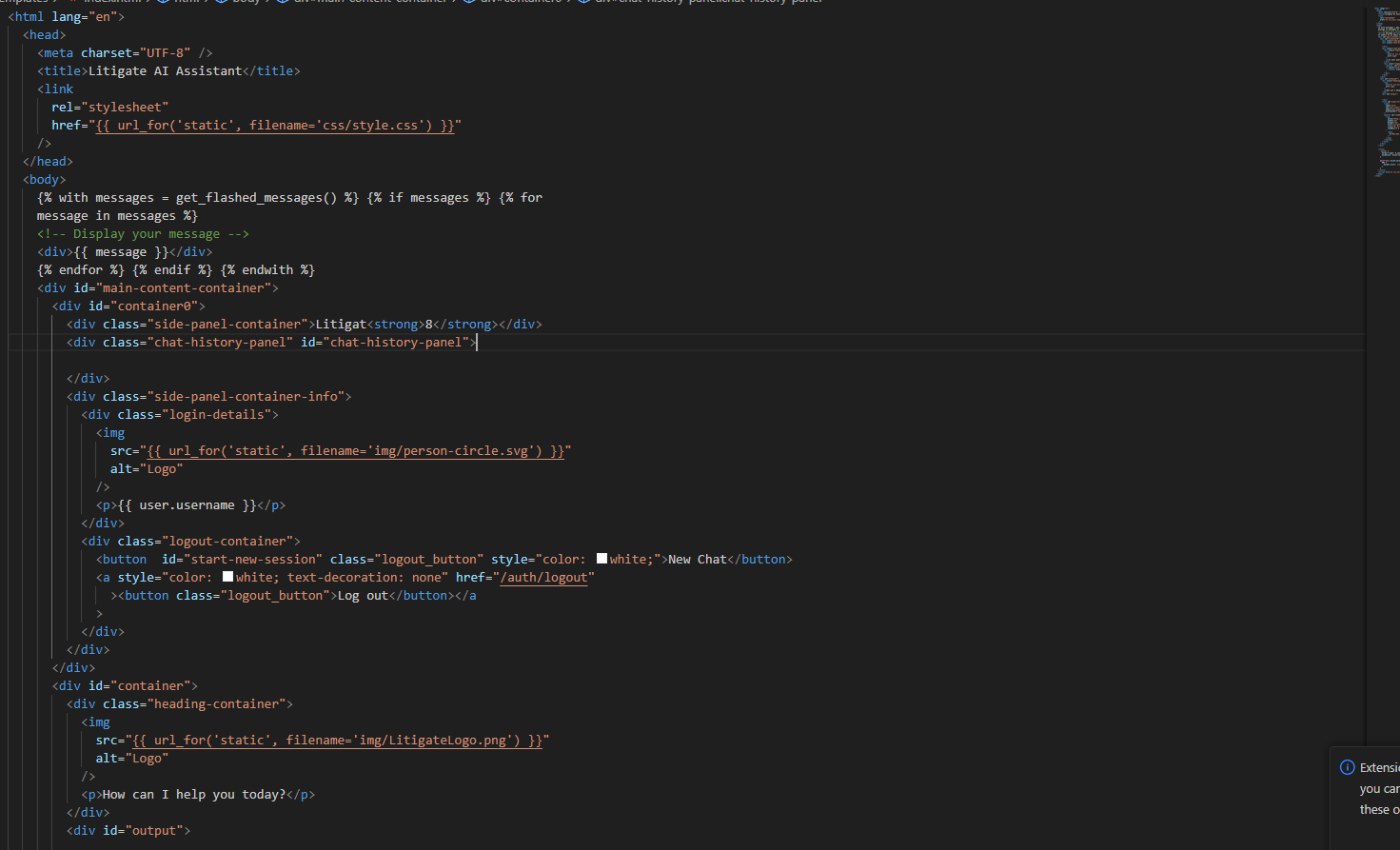


Figure 35 Showing the Front-end Implementation Chat Interface of Litigate

The main chat interface was build using combination of JavaScript, Html and CSS. The page followed the initial requirements set up by wireframes. With all the essential component present in the design. An input form to take in the User queries, a panel to display all the previous chat session with two buttons to either logout of the system or to start a new chat session.



This Marks the end of the front-end design implementation of the Litigat8. All the essential design elements have been put into place for the backend element to work on and make it dynamic. Moving to the next chapter the report would show the implementation of the backend of the system.

## Backend-Server Side:

For the implementation of the backend of the Litigat8 app various libraries were used such as **flask**, **SQLalchmey**, **functools**, **werkzeugsafe** etc. which worked in coordination to provide the implementation of the application based on the plans that were designed in the design phase of the project.

**Setting Up the App:**

The python flask application was set in the \_\_init\_\_.py file which would let you run the application using a simple **flask run** command in the CLI. The functionalities of the app was made modular and important functions stored in different files. All the necessary imports were made in the **\_\_init\_\_.py** file before setting up the app. The **creat\_app()** func sets up the application for us assimilating all the the routes and database configureation.

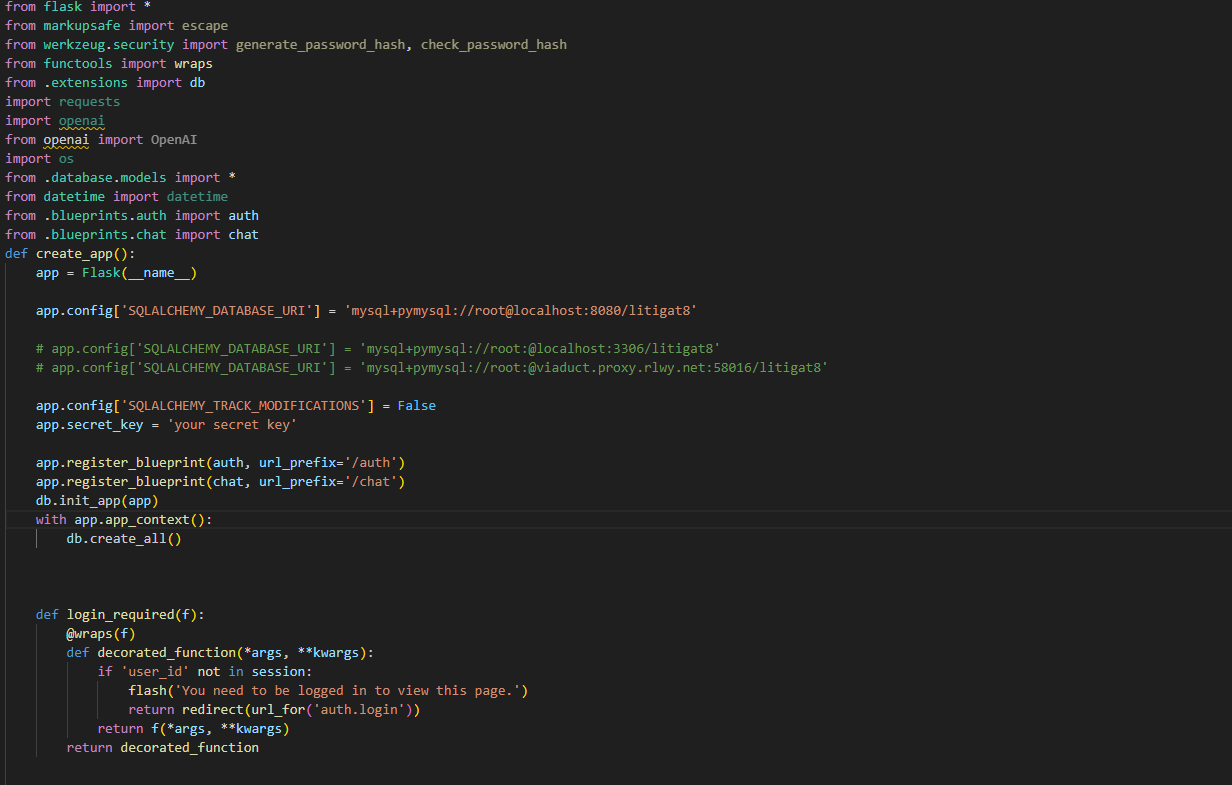


Figure 36 Showing the implementation of Flask App with Configuration

For setting up the routes for the flask application instead of setting it up directly in the app. Blueprints were used to set up the routes respective to their functionality. Two main blueprints were set **chat.py** and **auth.py** the **chat.py** handles all the routes to handle the server side functionality of chat functions linked with the specific routes. Whereas **auth.py** handles the logic used for authentication and registration of the user.

**Setting Up Blueprints:**

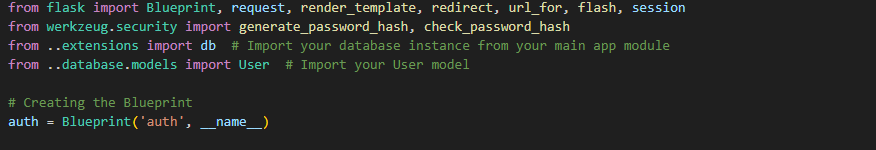


Figure 37 Showing the Implementation of Auth Blueprint

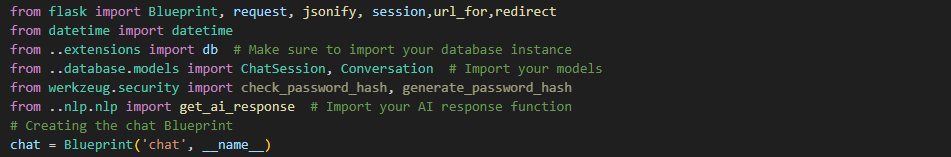


Figure 38 Showing the Implementation of Chat Blueprint

**Authentication Handling:**

The authentication of the User were done by using POST requests that are originated from the front-end. As a result of the POST request the function queries the database to check if the user exists or not and handles the results accordingly to the outcome.

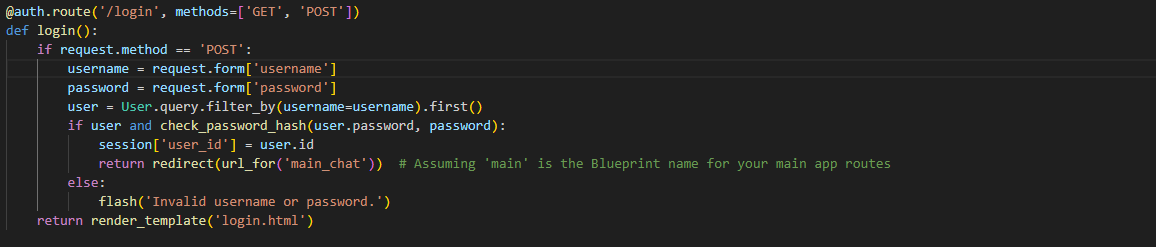


Figure 39 Showing the setting up of Login Function Server Point

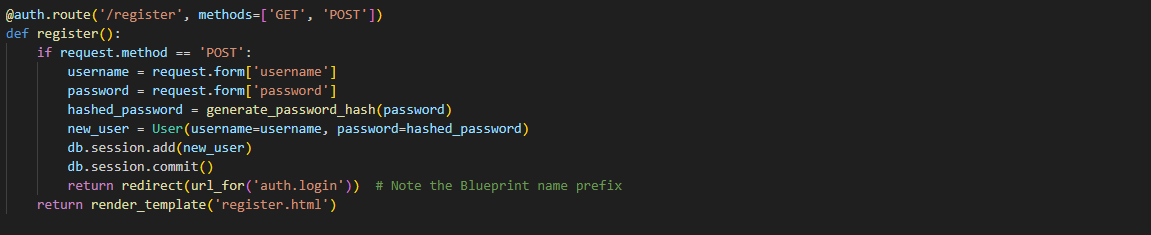


Figure 40 Showing the setting up of Register Function Server Point

Registration is done with a simple POST request as well to the URL **auth/register/**  which on being called inserts the User details into the database.

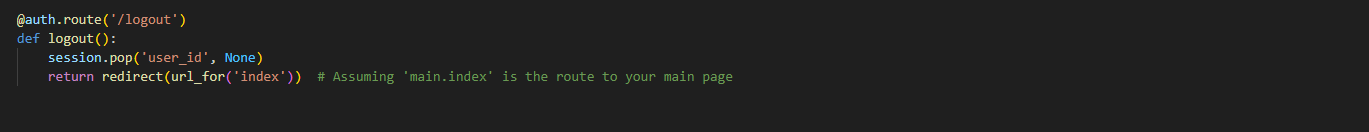


Figure 41Showing the setting up of Logout Function Server Point

The logout simply just pops out the user details if they exist in the flask application.

**Chat/Response Handling Generation and Chat Session Handling:**

**Python Server Side to Handle Requests:**

This section shows the implementation of how chat session are stored and retrieved. The **start\_chat\_session()** take in both kind of requests POST and GET. The function works by making Chat Session entry whenever the function is call and then the chat session id is stored in the session to be used later.

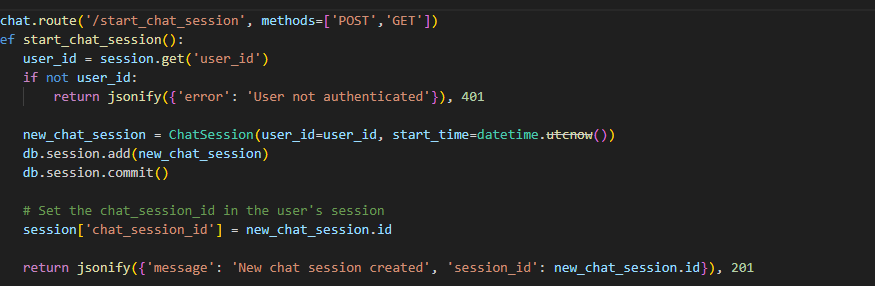


Figure 42 Showing the implementation of start\_chat\_session Server Point

The **/end\_chat\_session** works just by removing the chat\_session\_id from the session thus making not accessible anymore.

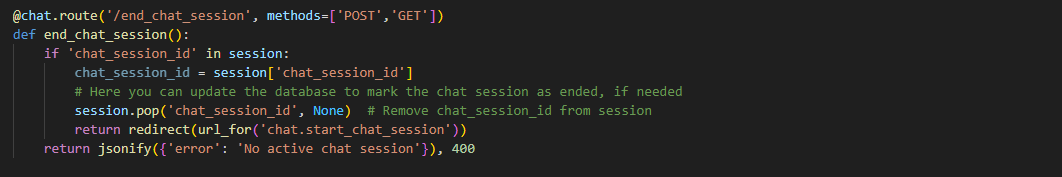


Figure 43 Showing the implementation of end\_chat\_session Server Point

The **/chat/submit** route is linked with **submit()**  function that calls onto the **ai\_response() func**  which is imported from **app.nlp.nlp\_engine** which serves as the base function for generating the AI response which then gets send to the font-end as JSON object to be printed out in a user understandable form. Using JSON to transfer data comes in really handy given the flexibility it provides.

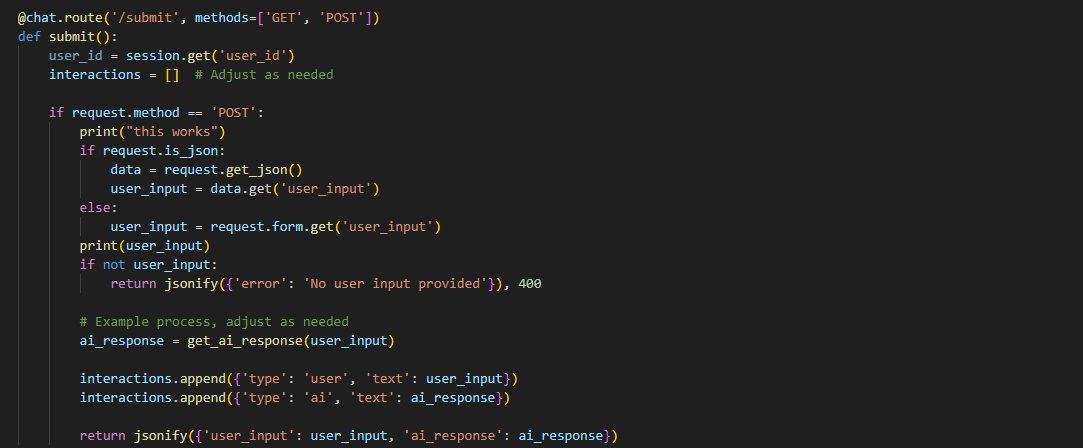


Figure 44 Showing the implementation of server handling of /submit route

The function **save\_interaction()** makes use of a helper function **save\_conversation \_message()** they both work to gather to save the interaction between the User and the AI to be referred later to be stored with a reference to the chat session. And the other function **get\_conversation/session\_id** was designed to get all the conversation stored in the database for a specific id. Which then are displayed on the front-end in a structured way.

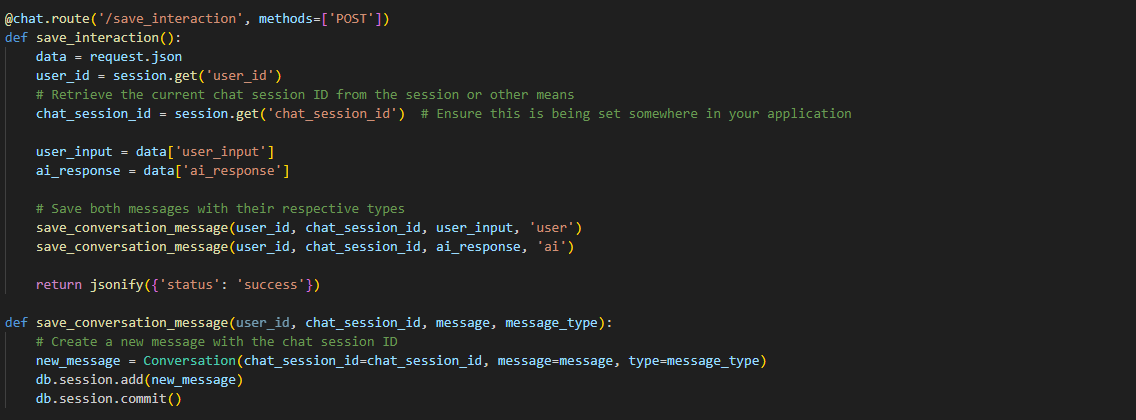


Figure 45 Showing the implementation of server handling of /save\_interaction route

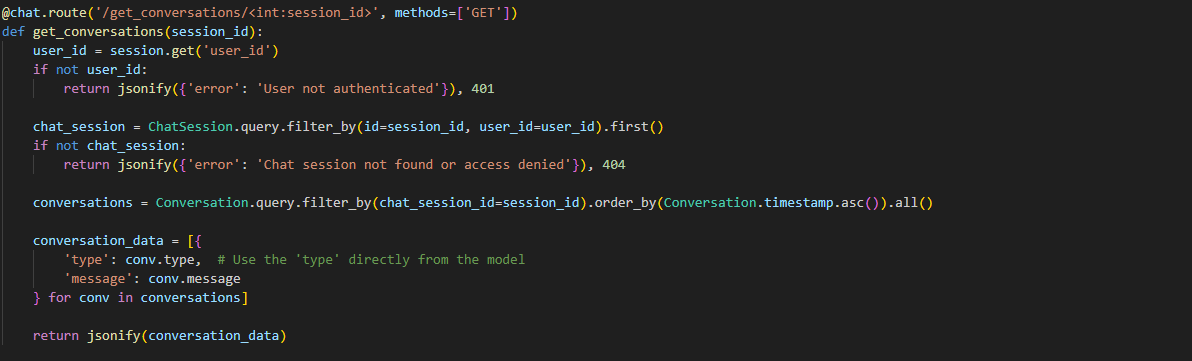


Figure 46 Showing the implementation of server handling of /get\_conversation/session\_id route

The loading of all the chat session to be displayed for the user was done by the use of **fetch\_chat\_histories()**  function which gets all the conversation enteries between the user and ai which have the same chat session id. It is then converted into a json object which then gets displayed at the front-end with the help of Javascript.



Figure 48 Showing the implementation of server handling of /fetch\_chat\_histories route

**JavaScript to Initiate fetch Requests to the Server Endpoint and Handle the JSON responses dynamically:**

This section marks the implementation of JavaScript that’s implemented in the front-end to which send fetch requests to server side to get the required data from the database and display it to the user.



Figure 49 Showing the implementation of Fetch Requests of chat/fetch\_chat\_history

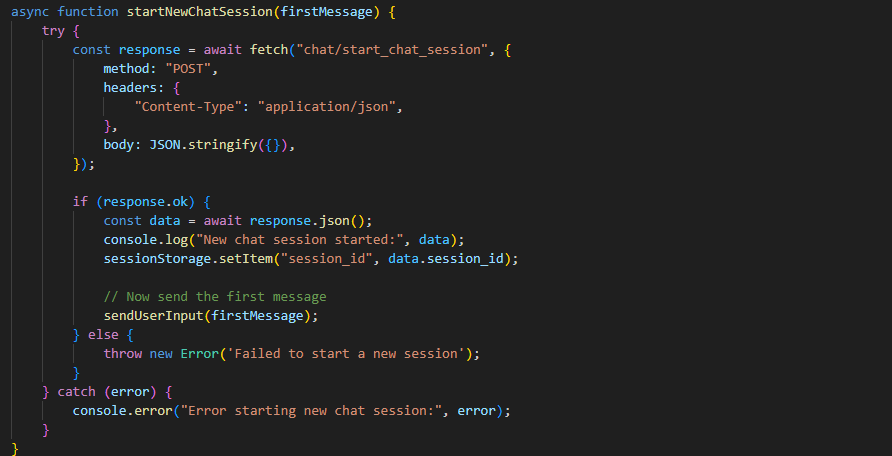


Figure 50 Showing the implementation of Fetch Requests of chat/start\_chat\_session



Figure 51 Showing the implementation of Fetch Requests of chat/submit

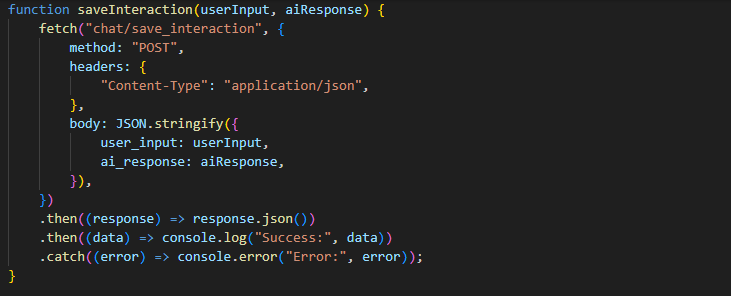


Figure 52 Showing the implementation of Fetch Requests of chat/save\_interaction

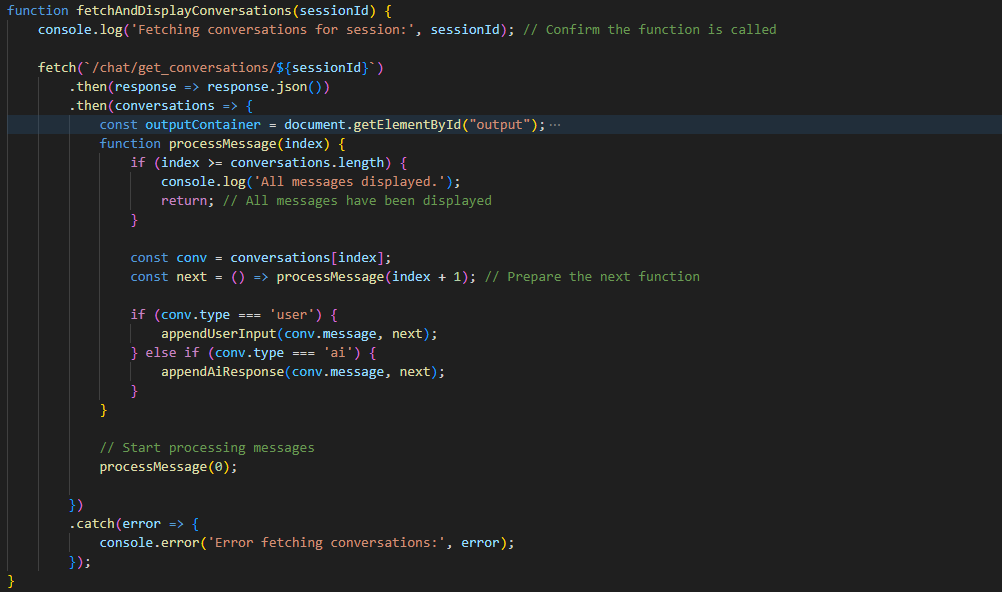


Figure 53 Showing the implementation of Fetch Requests of chat/get\_conversations/sessionID and Processing the of JSON load to front-end

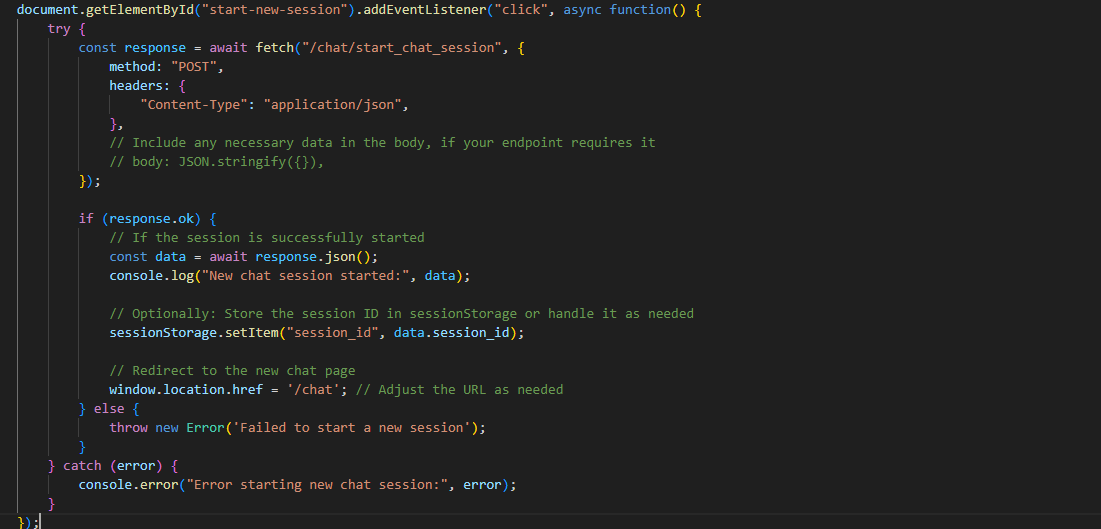


Figure 54 Showing the implementation of Fetch Requests of chat/start\_chat\_session

## NLP Model Implementation:

This chapter marks the implementation of the NLP Model. The steps that were taken to make a functioning LLM that would answer the User queries about household and tenant law would be discussed in this chapter. Many libraries were used to make the final proof-of-concept model for the application such as Langchain, OpenAI, Chroma, OpenAI, NLTK etc. These libraries served the foundation on which the application was build.

**Data Preprocessing/Text Preprocessing:**

Before starting with the development of the model, it was essential that the data is preprocessed before any work was done using it thus many preprocessing steps were taken to ensure there aren’t any discrepancies in the data. Four major preprocessing steps were taken , the first one being tokenization of the text. Which essentially breaks down the text into smaller units called tokens. There are different types of tokenization’s word tokenization, character tokenization etc. but in my case I decided to use word tokenization which is based on Penn Treeback tokenization because of the fact it can handle complex cases, such as contractions(e.g., splitting "don't" into "do" and "n't") and special punctuation patterns, thanks to its underlying use of regular expressions and the Penn Tree Bank tokenization standards.

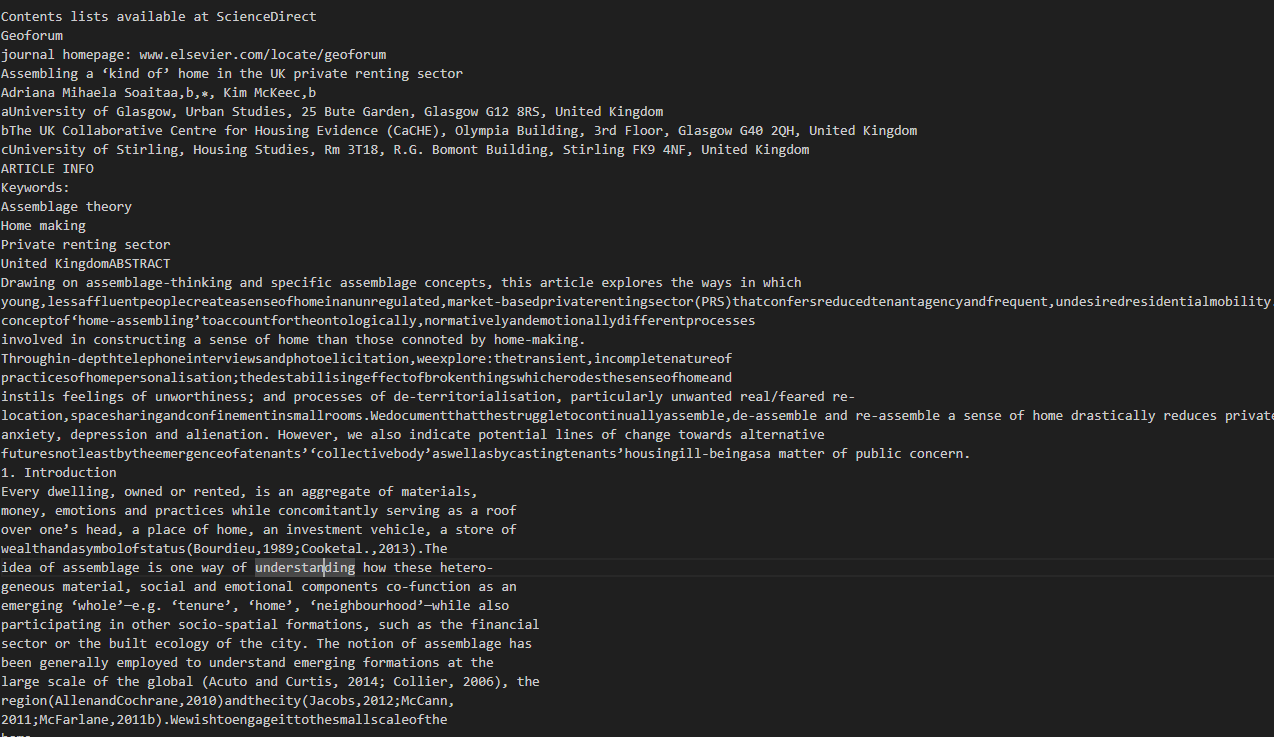
****

Figure 55 Showing the text data structure before pre-processing steps

The second pre-processing was lowercasing of all the words in a token. This made the whole text normalized which is essential to make sure there are not discrepancies in the corpus of the data.

The third pre-processing step that was taken was the removal of stop words in the case of English stop words like is,the,in don’t generally contribute to the meaning of a text for the analysis process thus they are removed from the dataset in the case of Litigat8 as well.



Figure 56 Showing the preprocessing steps i.e., Tokenization, Lowercasing, Removing Stop Words, Stemming

The 4th pre-processing step taken was stemming, which essentially breaks down the words to there root word for instance “Punctuation” would become “Punctuat” after applying stemming to it. There are number of benefits as it allowed to me to increase the relevance and search in information retrieval from the database.

****

Figure 57 Data Structure after Data Pre-Processing



Figure 58 Chunking/Spliting of the Text Data

The final step that was taken for the preprocessing was chunking of the data or splitting of the day of set size and of set character overlap. This help me increase the performance of the model as they have limit to input length of the tokens in case of the models that were used in Litigate from 512 to 4090. And the character overlapping helped me preserve the context of the tax and made sure it wasn’t lost.

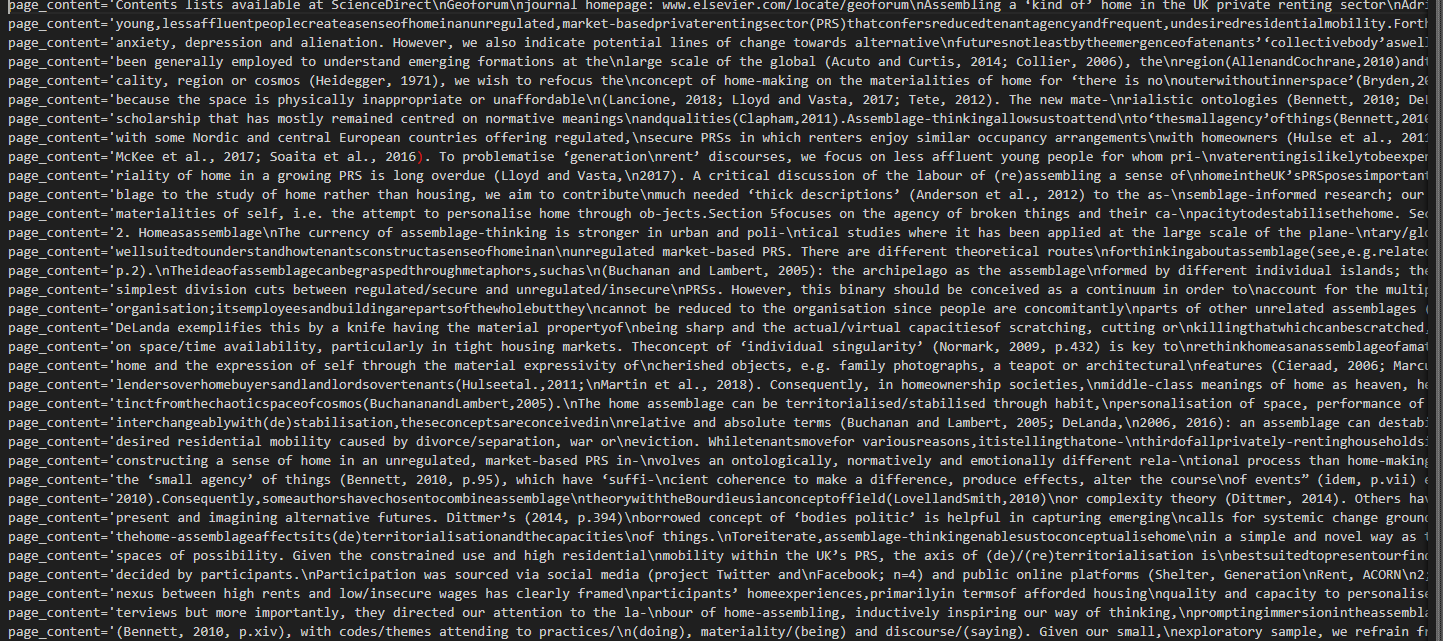


Figure 59 Data After Chunking/Splitting

**Embeddings Generation:**

The next step in sthe development was embeddings generation which are vector represetntation of text, words, phrases and entire documents. It maps the entities into a continuous multi-dimensional space where it reflects semantic relationships which exist in the space. There are number of reasons I employed this technique first was to establish semantic relationships between the words which improved my model’s similarity search functions. Which resulted in overall improvement of the model. Two different models were explored for making the embeddings, OpenAI embedding model **text-embedding-3-large** and hugging face **all-MiniLM-L6-v2,** they

Both were an excellent choice when generating the embeddings the only deciding factor to go with OpenAi was time it required to make the emdeddings which was relatively faster than hugging face model.



Figure 60 Showing the implementation of OpenAI Embedding Model

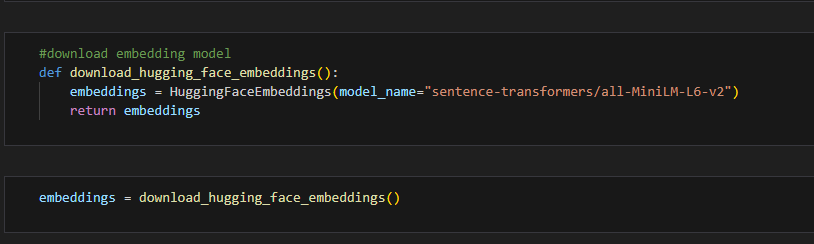


Figure 61 Showing the implementation of Hugging Face Embedding Model all-Mini-LM-L6-v2

**Vector Database Generation:**

The next step was to create vector database to store all the vectors that are created as a result of embeddings. The reason a vector database was employed was because of traditional database such as MySQL are not optimized to type of quereies machine learning applications require such as finding the nearest neighbors in a high dimensional space such as in the case of Litigat8. Setting up the vector database allowed me to do functions such as semantic search and dynamic content discovery.

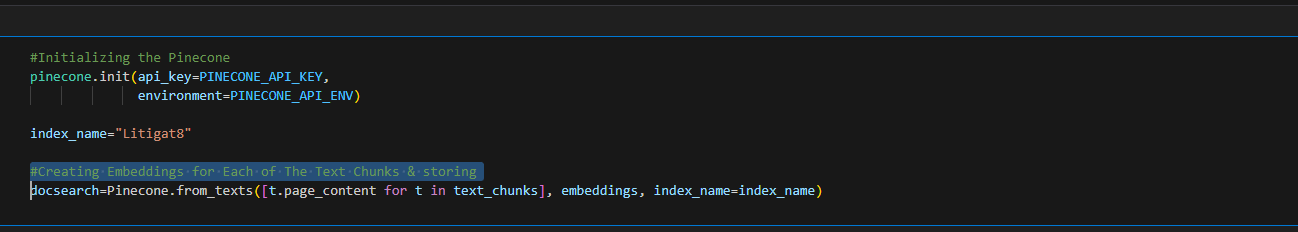


Figure 62 Showing the Implementation of Pinecone Vector Database

While making the vector database two different libraries were Analysed **Pinecone and Chroma**. Both of these libraries had there merits the Pincone database was an online vector database while chroma was a database on disk. Traditionally for larger project **Pinecone** would be the ideal choice and hence that option was explored first but due to recent development in the **Pinecone**  structure some of there methods and function have been depreciated while there were alternatives to that but the library that I was using for Langchain I was using wasn’t compatible for that. Ample time was given to make Pinecone work but due to not being enough resources available online. I wasn’t able to get it functioning with my model hence I decided to look for alternatives instead and thus decided to settle for an on disk database built using **Chroma.** Even though it might not be the ideal choice but for the scope of this application it provided enough value.

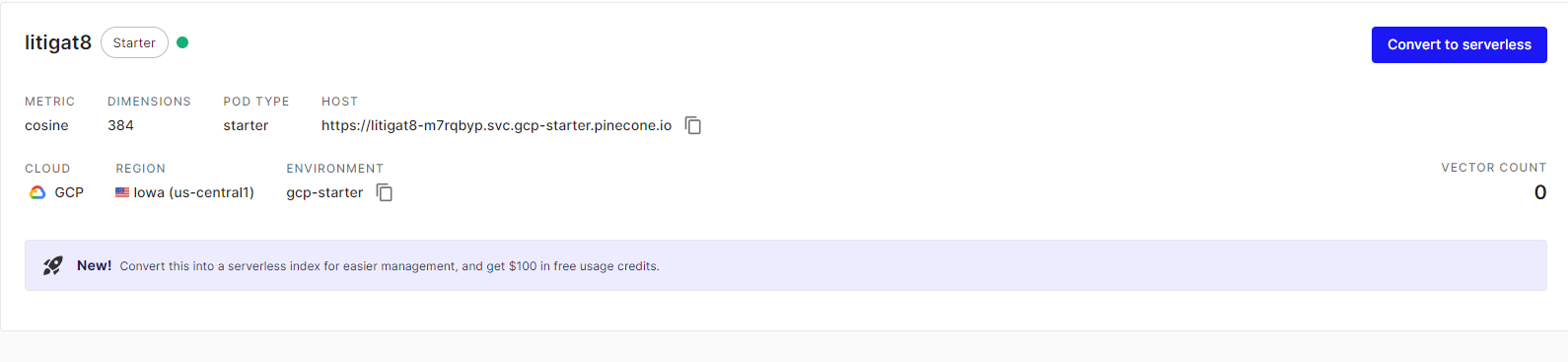


Figure 63 Showing the Setting Up of Pinecone Database on the Website

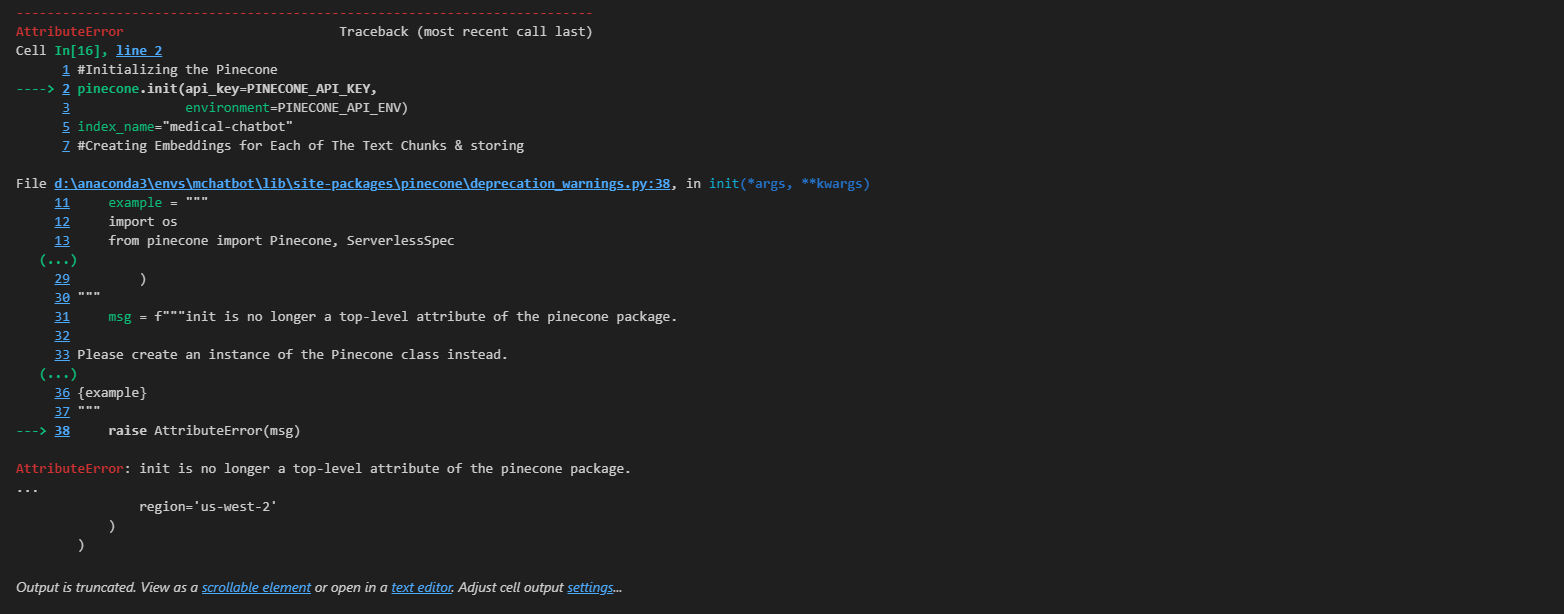


Figure 64 Showing the Depreciation Warning of Pinecone

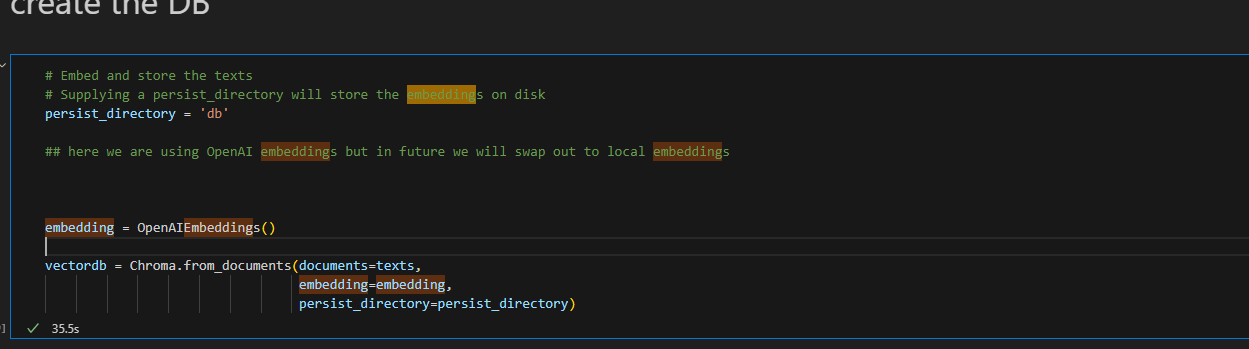


Figure 65 Showing the creation of Vector Database

**Making a Retriever:**

The retriever that was constructed from the database served as a way to query the database to get the specific piece of that that was required according to the query of the user.

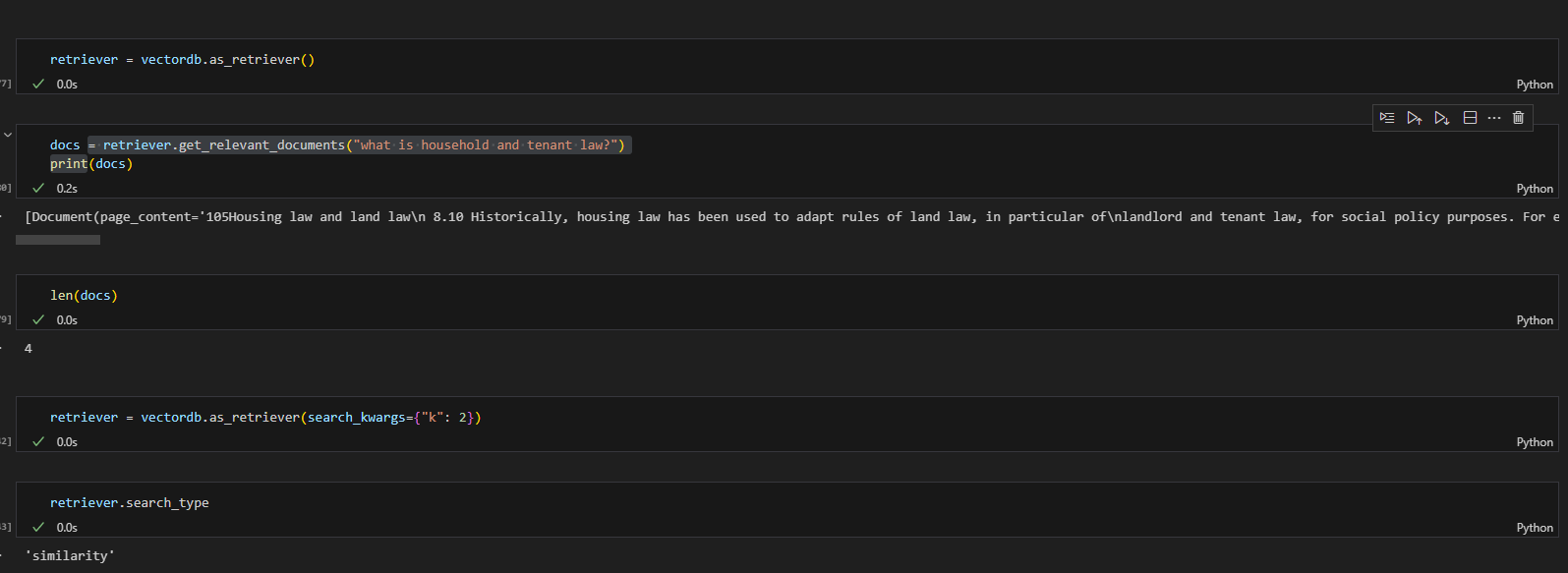


Figure 66 Showing the Implementation of VectorDb Retriever

**Making a Retrieval-based QA Chain for answering the question:**

The second last step in the development of the model was setting up the retrieval-based QA chain, which was done with the help of LangChain library. It helped me process the user question and understand key terms or entities. And enables me to retrieve information based on the processed user input and give that as a context to the model. And retriever was passed as a parameter as well which served as a way to get the information from the vector database.

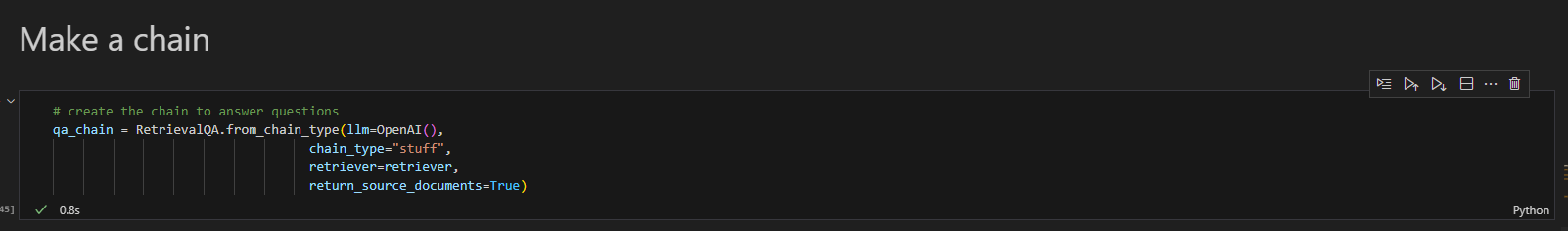


Figure 67 Showing the implementation of Retrieval-based QA Chain Using OpenAI() LLM

**Generating Reponses with Source Citation:**

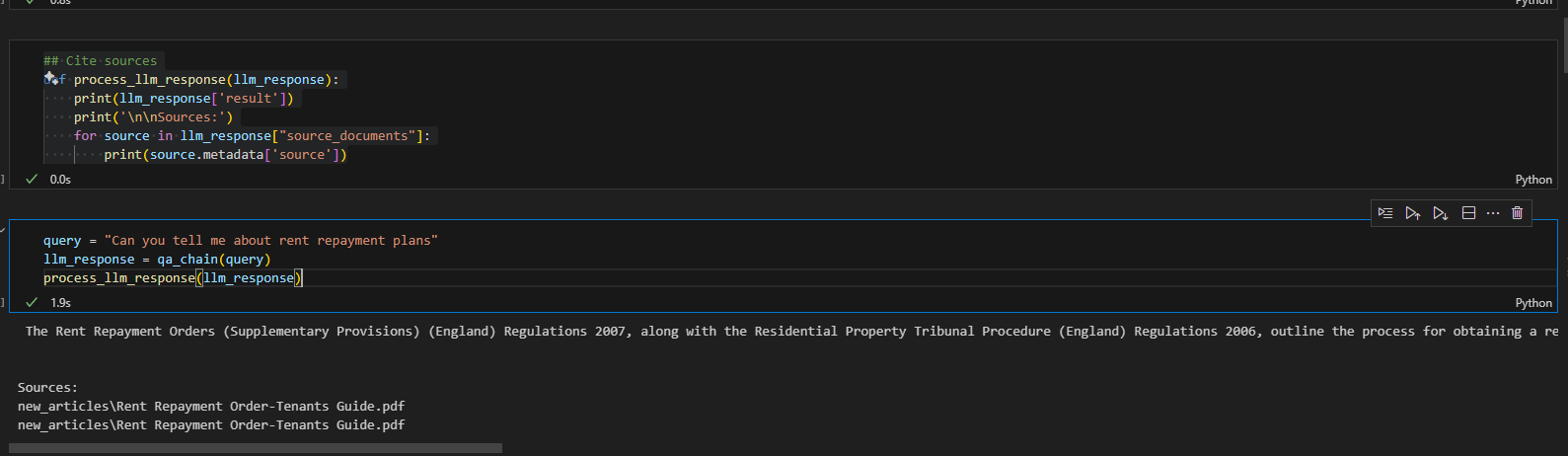


Figure 68 Showing the Response Generation with Source Citation

**Prompt Engineering:**

The final step of the model configuration was to set prompts for the Gen AI so that it doesn’t answer questions out of the context or it doesn’t answer question answer question which it doesn’t know answer for

**Making Sure the Generative AI doesn’t Answer Question Out of Context:**

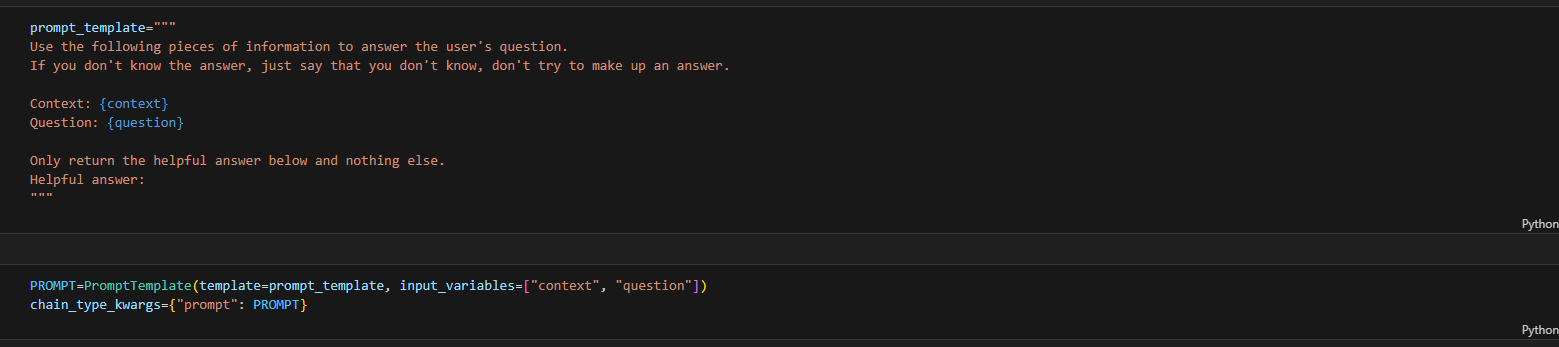


Figure 69 Setting up the context for the LLM

I played around the prompt settings to find the best prompt that would result in the best responses generated from the AI. Different versions of the prompts were created two of them are displayed in the Figure 71 and Figure 72.

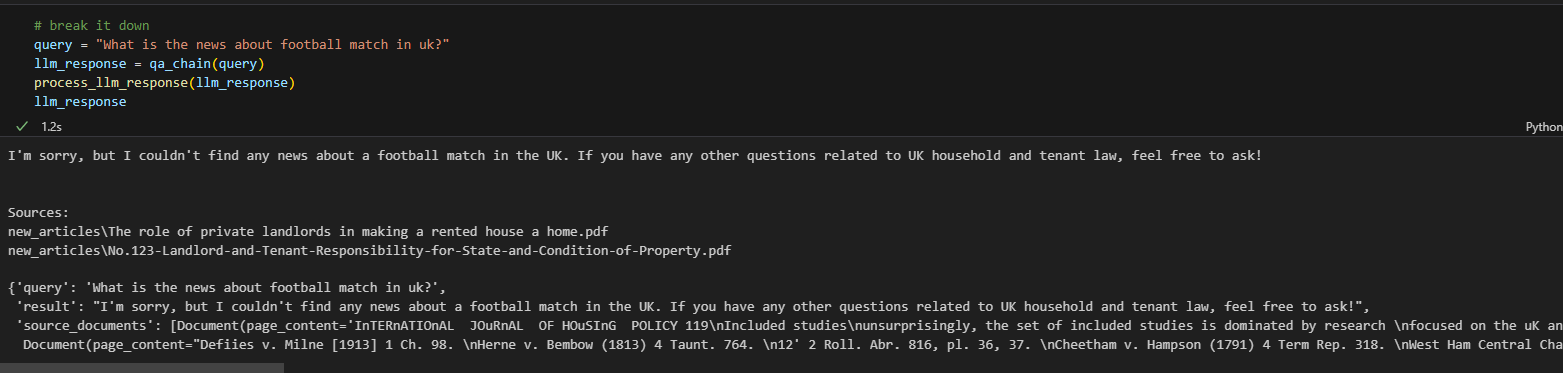


Figure 70 Showing the LLM answer to a Question out of Context

**Modifying the Prompt to get the Desired Outcome and Limiting the Gen AI:**

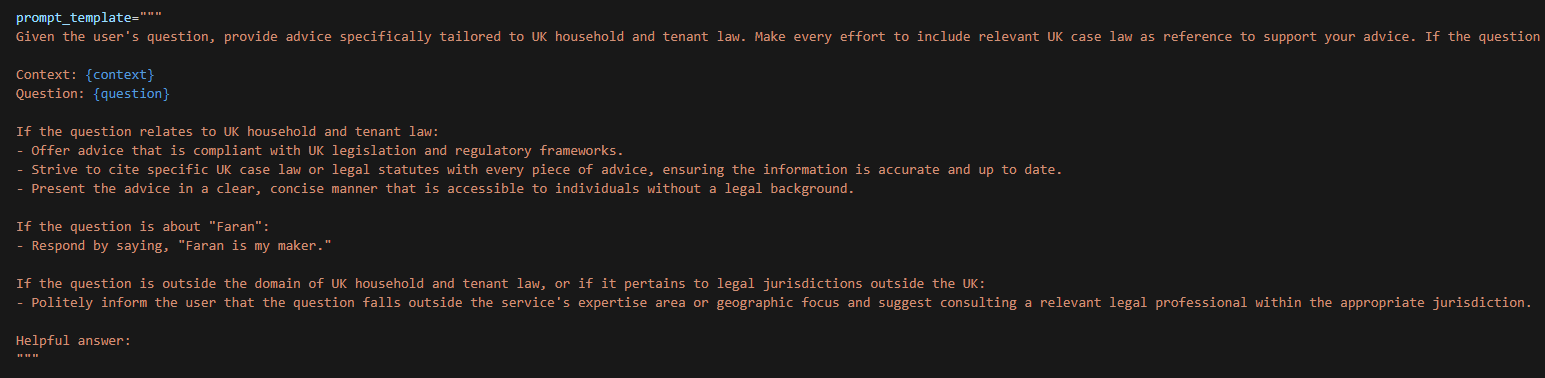


Figure 71 Prompt Engineering Version 1

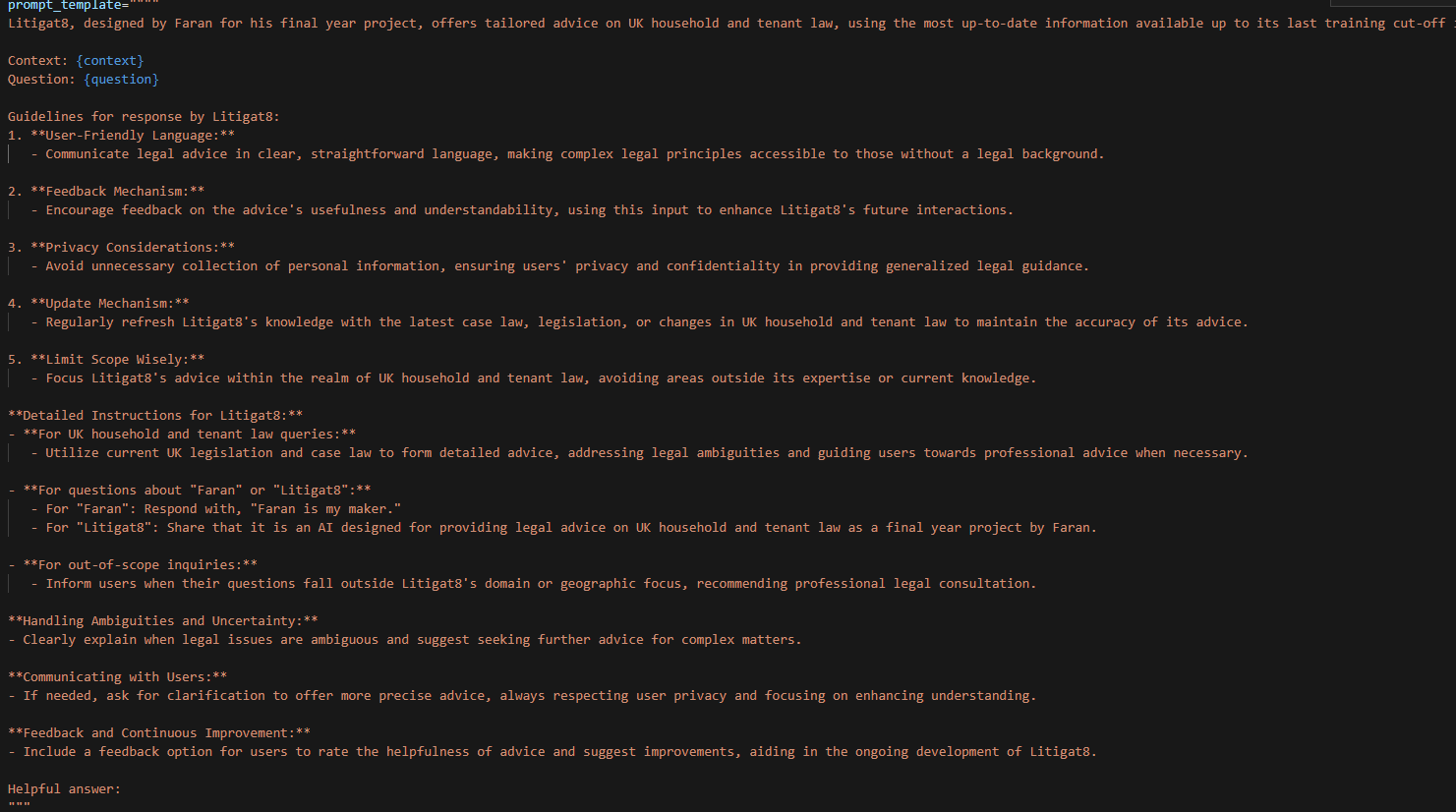


Figure 72 Prompt Engineering Version 2

**Comparison of ChatGPT 3.5 Turbo Vs ChatGPT 4.0 Turbo Vs Llama 2:**

The final step that was to be implemented was to do a comparative analysis between the LLM models that exits to finalize which would serve the function of litigate the best. The technical comparative analysis had two metrics which lead to the evaluation of the model the first one being the response time as one of the objectives were to display advices in real-time so it was of paramount the model should be able to generate responses according to that. The second metric was semantic similarity score of the response generate to figure out if the response generated has a meaning full context and semantic relation with the question that was asked by the user. In addition to that conversation coherence analysis was done was well which was carried out by myself to see how coherent the sentences are which model produces more meaningful and well structured sentences.

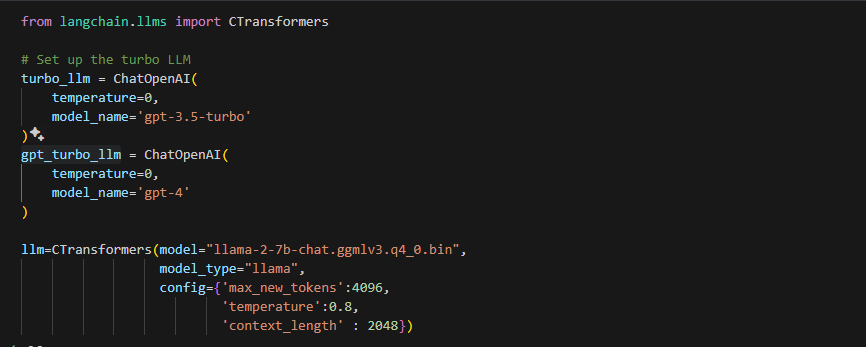


Figure showing the implementation of gpt-3.5-tubo , llama-2-7b Model , gpt-4.0

**Technical Analysis :Semantic Analysis and Response Time Comparison:**

The evaluation was carried out by setting up dummy questions and answer which were linked to specific prompts. And then the response generation time and the accuracy score (semantic score) were compared in the end for all three of the models.

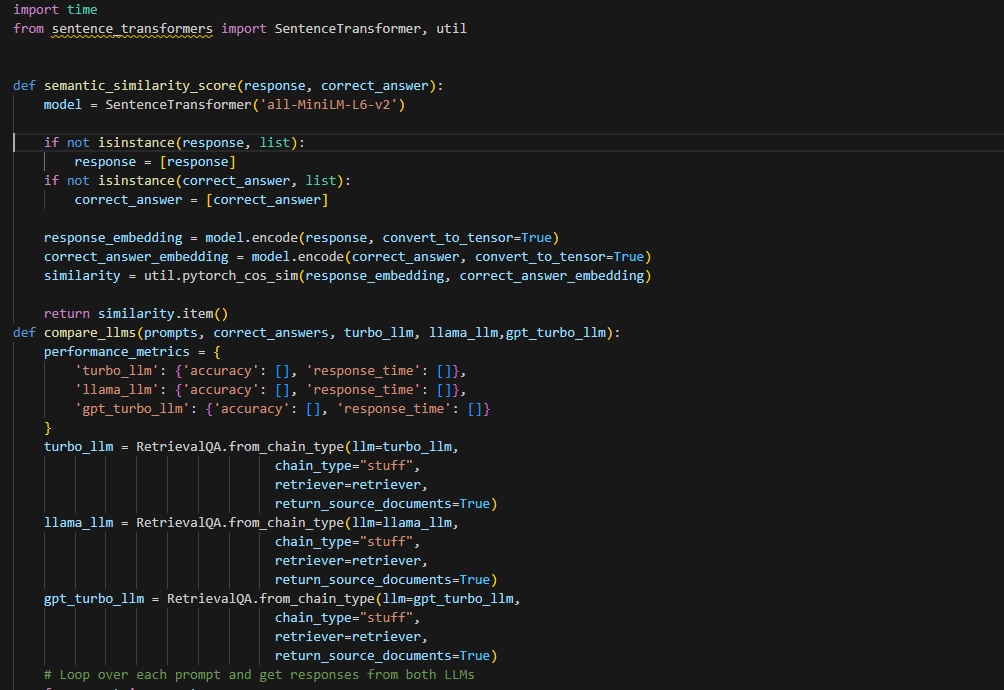


Figure 74 Setting Up the Semantic Similarity func and Comparison func

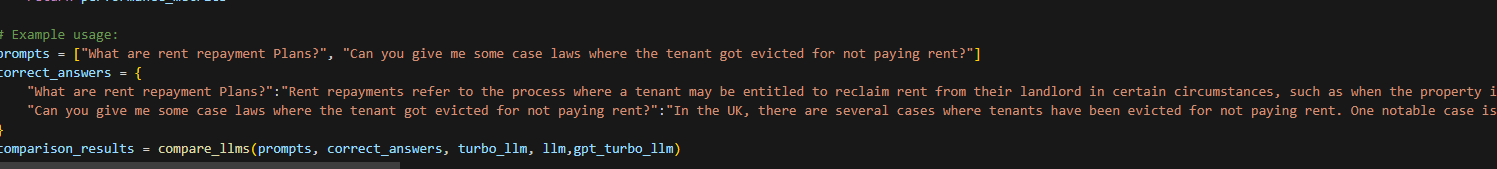


Figure 75 Setting up the Prompts and Correct Answers

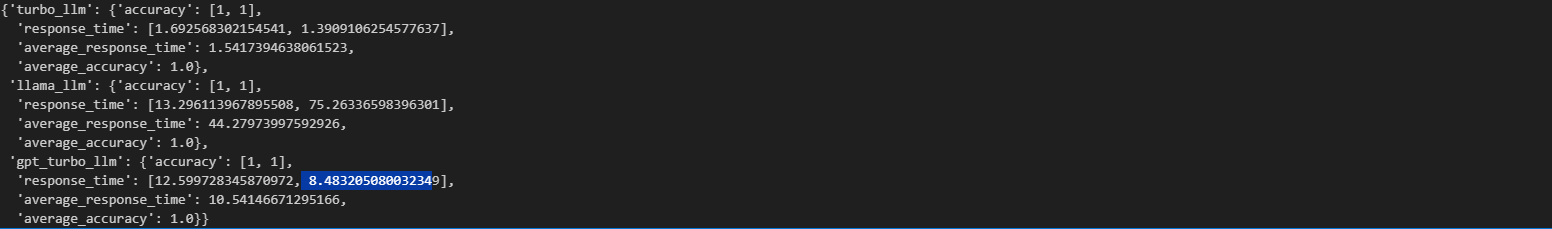


Figure 76 Showing the Comparison Result

**Conversational Coherence Analysis**

After the technical analysis, the models were compared on the bases of conversational coherence with a simple metric which response was more appealing to me as a user who’s seeking advice on a matter. For this purpose, a model question was decided on and was asked all three of the models and the responses were evaluated which would be discussed in the next chapter.

**Evaluation Sentence:** “What are rent repayments how can I pay them can you tell me?”

**LLAMA 2 Response time 2Min >**



Figure 77 Showing the response generated LLMA-2

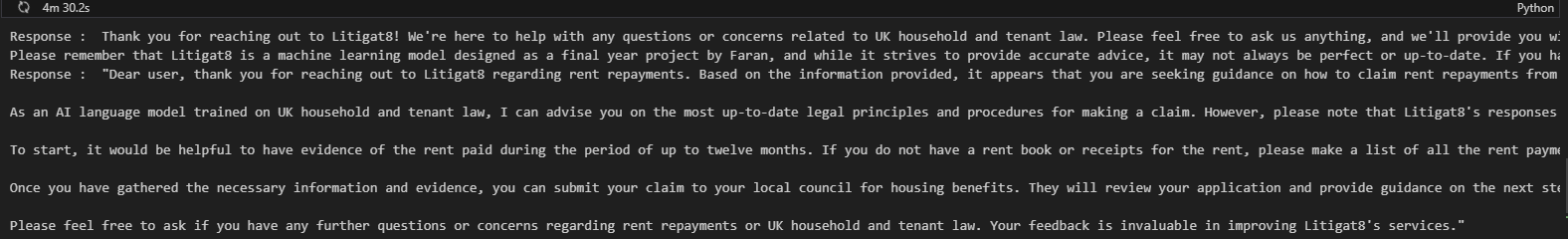


Figure 78 Showing the response generated LLMA-2

**GPT4 1Min >**

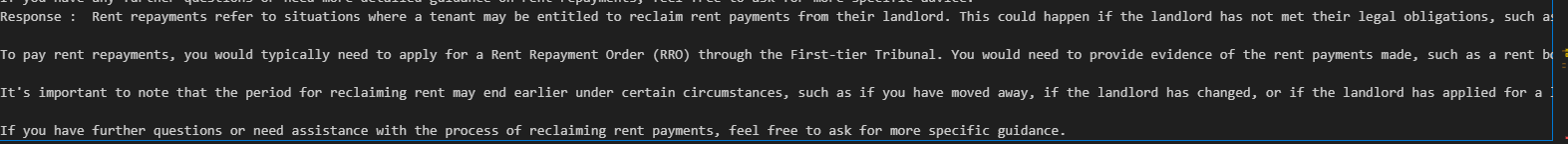


Figure 79 Showing the response Generated by gpt-4

**GPT 3.5 Turbo : 10 sec>:**



Figure 80 showing the response generated by gpt-3.

**Outcome Discussion**

Based on the results that were generated for the models **GPT-4**, **GPT-3.5-Turbo** and **LLAMA2**. When it comes to accuracy all three models performed as expected with having sematic relationship between the question asked and the response generated. Hence, they all passed on this evaluation. Moving on to the next evaluation, response time **GPT 3.5-turbo** was the fastest with response time ranging between 8 to 10 sec followed **by GPT-4** and then **LLAMA 2**. Hence **GPT 3.5-turbo** was the clear winner in the technical evaluation. Moving on to the Conversational Coherence Analysis **LLAMA2** and **GPT-4** provided with the best responses in the context of the data that was provided to them but **GPT-3.5-turbo** responses were accurate as well covering all the essential points to form a good response. As a result of all these observation GPT-3.5-turbo was selected as its stead true for all the metrics that were required for Litiagat8 to functions.