Pi Chat is a chatbot application that was created for the Pi DATACENTERS website. It is user-friendly and answers questions related to the company. It requires you to register(if you don't have an account) or sign in(once you've created your account). Only after doing so, can you access the chatbot. This application was made using Flask and SQLAlchemy for backend, HTML, CSS and JS for frontend.

**APP.PY**

This Flask application is a simple web-based chatbot platform that allows users to register, log in, and interact with a chatbot. Main components and functionalities of the program:

1. Import Statements: The program starts by importing necessary modules and classes from Flask, SQLAlchemy, Flask-Login, Flask-WTF, and other relevant libraries.
2. App Configuration: The Flask app is created using the Flask class. Various configurations are set, such as the database URI (SQLALCHEMY\_DATABASE\_URI), a secret key for security (SECRET\_KEY), and the path to the SQLite database file.
3. Database Setup: SQLAlchemy is used to define a database model called User, which represents registered users. It includes columns for the user's ID, username, and hashed password. The db instance is created to interact with the database.
4. User Authentication: Flask-Login is integrated for user authentication. The load\_user function is defined to retrieve a user based on their ID. The User class is set to inherit from UserMixin, which provides default implementations for user-related properties and methods.
5. Registration Form: A registration form class named RegisterForm is defined using Flask-WTF's FlaskForm. This form includes fields for username and password, with associated validators. Custom validation is added to ensure that the chosen username is unique.
6. Login Form: Similar to the registration form, a LoginForm class is defined. It includes fields for username and password, along with a custom authenticate method that checks the entered credentials against the stored user data.
7. App Routes:
   * 1. The '/' route maps to the home function, which renders the index.html template.
     2. The '/login' route handles user logins. If the form is submitted and validated, the user is redirected to the chatbot page after successful login. If login fails, an error message is displayed.
     3. The '/chatbot' route renders the base.html template, which contains the interface for interacting with the chatbot.
     4. The '/logout' route allows authenticated users to log out and redirects them to the home page.
     5. The '/register' route handles user registration. If the registration form is submitted and validated, the user's hashed password is stored in the database, and they are redirected to the login page.
8. Chatbot Interaction: The '/predict' route is a POST endpoint that accepts JSON data containing a user's message ("message"). The get\_response function (imported from chat.py, not included in the provided code) is expected to provide a response from the chatbot. The response is returned as JSON with the key "answer".
9. App Run: The \_\_name\_\_ condition ensures that the app is only run if the script is executed directly (not imported as a module). The app is run in debug mode on port 5802.

In summary, this Flask application provides a basic user authentication system, allowing users to register, log in, and interact with a chatbot. It utilizes Flask-WTF for form handling, Flask-Login for user authentication, and SQLAlchemy for database management. The provided code represents the foundation for a chatbot web application, with the actual chatbot logic missing (handled by the get\_response function).

**CHAT.PY**

This is a chatbot system that uses various NLP (Natural Language Processing) techniques to interact with users and provide informative responses. Main components and functionalities of the program:

1. Imports and Configuration:

- Various modules and classes are imported from the `langchain` package as well as external libraries such as `dotenv`, `constants`, and `pinecone`.

- The `.env` file is loaded to access configuration values, including the OpenAI API key and Pinecone environment information.

2. OpenAI API Setup:

- The OpenAI API key is set using the value from the configuration file.

- The `ChatOpenAI` instance (`llm`) is initialized, which seems to interact with the GPT-3.5 Turbo model for chat-based responses.

3. Templates and Prompt Setup:

- A system message template (`system\_msg\_template`) and a human message template (`human\_msg\_template`) are defined to structure the conversation.

- A chat prompt template (`prompt\_template`) is constructed using these templates to guide the conversation.

4. Conversation Chain Setup:

- A `ConversationChain` instance (`conversation`) is created. It combines the chat prompt template, an NLP model (`llm`), and a conversation memory (`buffer\_memory`) to manage the chatbot's conversational context.

5. Embeddings and Pinecone Index Setup:

- Embeddings (vector representations) are used to represent text data. OpenAI embeddings (`OpenAIEmbeddings`) are initialized.

- A Pinecone index is initialized based on the embeddings.

6. Query Refinement:

- The `query\_refiner` function refines user queries to generate more relevant questions based on the conversation history and the user's query. The OpenAI API is used to achieve this.

7. Get Conversation String:

- The `get\_conversation\_string` function constructs a formatted string representation of the conversation history (human messages and bot responses).

8. Response Generation:

- The `get\_response` function generates a response from the chatbot based on user input.

- The conversation history and user's query are used to refine the query using the `query\_refiner`.

- The Pinecone index is searched for similar documents based on the refined query.

- The `conversation` chain is used to predict a response using the context from the similar documents and the user's query.

In summary, this script showcases the development of a chatbot system that employs OpenAI's GPT-3.5 Turbo model for generating responses. It uses conversation templates, embeddings, and Pinecone indexing to refine queries and provide more contextually relevant responses. The script demonstrates the interaction between different NLP components to create a chatbot capable of engaging in meaningful conversations with users.

**UTILS.PY**

Utils.py performs a series of tasks to retrieve data from a website, split the data into smaller chunks, generate embeddings for the text chunks, and push these embeddings to the Pinecone service for indexing and retrieval. It is meant to be run only once to initially populate the database with the processed data. Let's break down the code step by step:

1. Importing Required Modules:

The script starts by importing various modules and classes necessary for its functionality. These include modules for text splitting, vector storage, embeddings generation, Pinecone interaction, asyncio, and more.

2. Loading Environment Variables:

The 'dotenv' module is used to load environment variables from a '.env' file. This is a common practice to store sensitive information like API keys and configuration parameters outside of the codebase.

3. Defining `get\_website\_data` Function:

This function is responsible for fetching data from a website using the SitemapLoader. The SitemapLoader seems to be part of the 'langchain' library and is used to load documents from a given website's sitemap.

4. Defining `split\_data` Function:

This function utilizes the RecursiveCharacterTextSplitter to split the loaded documents into smaller chunks. The text splitter seems to break the documents into chunks based on specified parameters such as 'chunk\_size' and 'chunk\_overlap'.

5. Defining `create\_embeddings` Function:

This function is used to create an embeddings instance, which is essentially a mechanism to convert text data into numerical vectors that represent the semantics of the text. The code uses either the 'SentenceTransformerEmbeddings' or the 'OpenAIEmbeddings' class to generate these embeddings.

6. Defining `push\_to\_pinecone` Function:

This function interacts with the Pinecone service to initialize a connection and push the generated embeddings to an index. Pinecone is a vector database that allows fast similarity search and retrieval based on vector representations. The Pinecone index is created from the embeddings and associated with a specified environment and index name.

7. Fetching Data and Processing:

The script initializes by fetching data from a specified website URL using the `get\_website\_data` function. It then prints a message indicating that the data has been retrieved. The retrieved documents are split into smaller chunks using the `split\_data` function, and a message is printed to indicate that the data has been split.

8. Generating Embeddings:

Embeddings are generated using the `create\_embeddings` function, and a message is printed to indicate that the embeddings have been created.

9. Pushing to Pinecone:

Finally, the script initializes a connection with the Pinecone service and pushes the generated embeddings to the Pinecone index using the `push\_to\_pinecone` function. A message is printed to indicate that the data has been pushed to Pinecone.

**CONSTANTS.PY**

These constants hold specific values that are relevant to the functionality of the broader codebase.

**.ENV**

The .env file is a plain text file that contains a set of key-value pairs, each representing an environment variable and its associated value. This file is meant to be kept private and not shared publicly, as it often contains sensitive information that is specific to the deployment environment. It is used to store the OpenAI API key.

**.ENV.EXAMPLE**

The .env.example file is a template or example of what the .env file should contain. This file is often included in the code repository and shared publicly. It serves as a reference for developers, indicating which environment variables are expected and what kind of values they should hold.

**TEMPLATES:**

**INDEX.HTML**

This file renders the home page of the application. The user is directed to select ‘LOGIN’ or ‘REGISTER’ option. On clicking login button, they will be redirected to the ‘LOGIN’ page and similarly for the ‘REGISTRATION’ page. Styling is done using CSS.

**LOGIN.HTML**

Login page allows already existing users to log into their accounts and start using the Pi chatbot. If username or password is invalid, an error message is displayed. Styling is done using CSS.

**REGISTER.HTML**

Registration page enables users to sign up and then access the chatbot. If username is already taken, an error message is displayed. Styling is done using CSS.

**BASE.HTML**

This file renders the chatbot page. First, a successful login message is displayed to the user. The chatbot can now be accessed by the user and user can ask any questions they have about Pi. Styling is done using CSS and JavaScript.

**STATIC:**

**APP.JS**

This JS file is used to add responsivity and functionality to our chatbot. It uses event listeners to understand the user’s requirements. It displays the new messages on the chatbot and also shows all the previous messages.

**HOME.CSS, LOGIN.CSS, STYLE.CSS**

Responsible for styling the elements.