**Project Report: Chatbot Application**

**1. Introduction**

This project aims to build a fully functional chatbot application that allows users to log in, interact with a chatbot, and retrieve information from a backend API. The application integrates OpenAI's API for conversational responses, uses Flask for backend API development, and ReactJS for frontend development.

**2. Technology Stack**

* **Frontend**: ReactJS
  + **React Hooks**: Used for managing state and side effects in functional components.
  + **Axios**: For making HTTP requests from the frontend to the backend API.
* **Backend**: Flask (Python)
  + **Flask-JWT-Extended**: Used for handling user authentication via JWT tokens.
  + **OpenAI API**: Provides conversational responses through the OpenAI language model.
  + **SQLite**: For storing mock product data and enabling basic queries for the chatbot.
* **Authentication**: JWT (JSON Web Tokens) for secure login and authorization.

**3. Project Setup**

**Frontend Setup:**

1. Install Node.js and npm.
2. Install required dependencies using:

bash

Copy code

npm install

1. Start the React development server:

bash

Copy code

npm start

The frontend will be available at http://localhost:3000.

**Backend Setup:**

1. Install Python and pip.
2. Install dependencies using:

bash

Copy code

pip install -r requirements.txt

1. Create a .env file with necessary API keys and JWT secrets:

bash

Copy code

OPENAI\_API\_KEY=your\_openai\_api\_key

JWT\_SECRET\_KEY=your\_jwt\_secret\_key

1. Run the backend:

bash

Copy code

python app.py

The backend will be available at http://127.0.0.1:5000.

**4. Sample Queries and Results**

The chatbot can respond to user queries based on product categories (from mock data). Below are some sample queries and the corresponding results obtained:

**Sample Query 1:**

* **Query**: "Smart Watches"
* **Response**:

json

Copy code

[

{

"category": "Electronics",

"id": 1,

"name": "Smart Watch",

"price": 199.99,

"stock": 50

}

]

**Sample Query 2:**

* **Query**: "Cotton Shirts"
* **Response**:

json

Copy code

[

{

"category": "Textiles",

"id": 2,

"name": "Cotton Shirt",

"price": 25,

"stock": 75

}

]

**Sample Query 3:**

* **Query**: "Books"
* **Response**:

json

Copy code

[

{

"category": "Books",

"id": 3,

"name": "Novel XYZ",

"price": 15,

"stock": 100

}

]

**5. Project Architecture**

* **Frontend**:
  + ReactJS handles the user interface. It allows users to log in, type queries, and view responses in real-time.
  + Axios is used for sending queries to the Flask backend and receiving responses.
* **Backend**:
  + Flask serves as the API that processes the user's queries and interacts with the OpenAI API.
  + The backend also handles JWT-based authentication and returns mock data about products when requested.

**6. Key Learnings**

* **Asynchronous Operations**: Managing asynchronous API calls with Axios in React was a key learning. The interaction between the frontend and backend required understanding asynchronous behavior and managing the UI state effectively.
* **JWT Authentication**: Implementing JWT tokens for user authentication provided a robust way to secure the application. Understanding how to generate, store, and validate JWT tokens was crucial.
* **OpenAI API**: Integrating with the OpenAI API provided hands-on experience in working with external services for natural language processing, and it enhanced the chatbot’s conversational abilities.
* **Modular Codebase**: By separating concerns between frontend (ReactJS) and backend (Flask), the codebase was kept modular and scalable. This design makes it easier to add new features or modify existing ones.

**7. Potential Challenges and Solutions**

* **Error Handling**: Handling errors gracefully was challenging, especially dealing with network issues and API errors. We implemented robust error handling both on the frontend (ReactJS) and backend (Flask) to ensure the application does not crash.
* **Authentication Flow**: The user authentication flow with JWT tokens required careful management of the authentication state on the frontend and ensuring that tokens were properly validated on the backend.
* **API Integration**: Integrating OpenAI’s API was challenging, especially when handling different response formats. We made sure that the API returned useful and accurate responses by properly formatting the requests and responses.