**Fintech LLM Chatbot – Detailed Documentation**

**1. Introduction**

The Fintech LLM Chatbot is a production‑grade, multi‑tier system designed to handle real‑time financial queries in a secure and scalable way. The project combines a modern React-based frontend with FastAPI microservices on the backend and integrates a fine‑tuned large language model (LLM) for natural language processing. In this document, we explain all aspects of the task implemented so far, including the overall architecture, individual components, integration strategies, and key design decisions.

**2. Overall System Architecture**

The system is divided into layers:

* **Client (Frontend)**  
  A React application (using TypeScript, Redux, and Material‑UI) running on **http://localhost:3000** provides the user interface where users can log in, submit queries, and view responses.
* **Backend Microservices**  
  Two main FastAPI microservices are implemented:
  + **User Management Service (Port 8000):** Responsible for user authentication, secure login, and JWT token generation.
  + **Query Processing Service (Port 8001):** Handles incoming financial queries, verifies authentication, and processes queries via an integrated LLM component.
* **LLM Integration (Llama 3 Simulation):**  
  A dedicated module uses Hugging Face’s Transformers library to simulate a Llama 3 model (initially via a dummy function). This module processes user queries and generates responses.
* **Middleware and Security**  
  Both microservices have CORSMiddleware enabled to allow cross‑origin requests from the frontend. OAuth2 and JWT mechanisms are used for securing the endpoints.

A detailed architecture diagram is saved in the **docs/architecture.drawio.png** file. This diagram visually represents the interactions between the React app and the backend services along with all supporting components.

**3. Detailed Component Breakdown**

**3.1 Frontend (React Application)**

**Technologies Used:**

* React with TypeScript
* Redux for state management
* Material‑UI for UI components
* Axios for HTTP requests

**Functionality:**

* **Login Screen:**  
  Users enter their credentials (using dummy credentials such as **test@example.com** and **test123**). The login component (Login.tsx) sends an HTTP POST request with form‑encoded data to the User Management microservice (**http://localhost:8000/user/token**). Upon successful login, the JWT token is stored in Redux.
* **Dashboard:**  
  Once logged in, the Dashboard component allows users to enter a financial query. The query (for example, “tell about apple stocks”) is sent as JSON via an axios POST request to the Query Processing service (**http://localhost:8001/query/process\_query**).  
  The dashboard then displays the answer returned from the backend.

**Key Code Snippets:**

* *Redux Store Setup (store.ts):*

JAVASCRIPT

1import { configureStore } from '@reduxjs/toolkit';

2import userReducer from '../features/user/userSlice';

3

4export const store = configureStore({

5 reducer: { user: userReducer },

6});

* *Login Component (Login.tsx):*

JAVASCRIPT

Collapse

1import React, { useState } from "react";

2import { useDispatch } from "react-redux";

3import { setCredentials } from "../features/user/userSlice";

4import { TextField, Button, Container, Typography } from "@material-ui/core";

5import axios from "axios";

6import qs from "qs";

7

8const Login: React.FC = () => {

9 const dispatch = useDispatch();

10 const [email, setEmail] = useState("");

11 const [password, setPassword] = useState("");

12

13 const handleLogin = async () => {

14 try {

15 const data = qs.stringify({ username: email, password });

16 const response = await axios.post("http://localhost:8000/user/token", data, {

17 headers: { "Content-Type": "application/x-www-form-urlencoded" },

18 });

19 dispatch(setCredentials({ token: response.data.access\_token, username: email }));

20 alert("Login successful!");

21 } catch (error: any) {

22 console.error("Login failed:", error);

23 alert("Error logging in: " + (error.response ? error.response.data.detail : error.message));

24 }

25 };

26

27 return (

28 <Container maxWidth="sm">

29 <Typography variant="h4">Login</Typography>

30 <TextField label="Email" fullWidth value={email} onChange={(e) => setEmail(e.target.value)} />

31 <TextField label="Password" type="password" fullWidth value={password} onChange={(e) => setPassword(e.target.value)} />

32 <Button variant="contained" color="primary" onClick={handleLogin}>Log In</Button>

33 </Container>

34 );

35};

36

37export default Login;

* *Query Submission in Dashboard (Dashboard.tsx):*

JAVASCRIPT

Collapse

1import React, { useState } from "react";

2import { Container, Typography, Button, TextField } from "@material-ui/core";

3import axios from "axios";

4import { useSelector } from 'react-redux';

5import { RootState } from '../app/store';

6

7const Dashboard: React.FC = () => {

8 const [query, setQuery] = useState("");

9 const [result, setResult] = useState("");

10 const token = useSelector((state: RootState) => state.user.token);

11

12 const handleQuery = async () => {

13 try {

14 const response = await axios.post(

15 "http://localhost:8001/query/process\_query",

16 { query },

17 { headers: { "Content-Type": "application/json", Authorization: "Bearer " + token } }

18 );

19 console.log("Response from server:", response.data);

20 setResult(response.data.answer);

21 } catch (error: any) {

22 console.error("Error processing query:", error);

23 setResult("Error processing query");

24 }

25 };

26

27 return (

28 <Container maxWidth="sm">

29 <Typography variant="h4">Financial Query Dashboard</Typography>

30 <TextField label="Enter your financial query" fullWidth value={query} onChange={(e) => setQuery(e.target.value)} />

31 <Button variant="contained" color="primary" onClick={handleQuery}>Submit Query</Button>

32 {result && <Typography variant="body1">{result}</Typography>}

33 </Container>

34 );

35};

36

37export default Dashboard;

**3.2 User Management Microservice**

**Technologies Used:**

* FastAPI in Python
* JWT for authentication (with python-jose or PyJWT)
* OAuth2PasswordBearer for handling credentials
* CORSMiddleware for CORS

**Functionality:**

* **Authentication Endpoint:**  
  The service exposes a POST endpoint **/user/token** which accepts form‑encoded data (username and password). It verifies the credentials against a dummy in‑memory user database.
* **JWT Generation:**  
  On successful authentication, a JWT token is generated using a secret key. This token is later used by the Query Processing service to authenticate requests.
* **CORS Handling:**  
  CORSMiddleware is configured to accept requests originating from **http://localhost:3000**.

**Key Code Snippet (User Management – main.py):**

Python

Collapse

1from fastapi import FastAPI, HTTPException, Depends

2from fastapi.middleware.cors import CORSMiddleware

3from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

4from datetime import datetime, timedelta

5import jwt

6

7app = FastAPI(title="User Management Service")

8

9app.add\_middleware(

10 CORSMiddleware,

11 allow\_origins=["http://localhost:3000"],

12 allow\_credentials=True,

13 allow\_methods=["\*"],

14 allow\_headers=["\*"],

15)

16

17SECRET\_KEY = "your-secret-key" # Replace with a secure secret or environment variable

18oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="/user/token")

19

20users\_db = {

21 "test@example.com": {"password": "test123", "username": "test"}

22}

23

24def create\_access\_token(data: dict, expires\_delta: timedelta = timedelta(minutes=30)):

25 to\_encode = data.copy()

26 expire = datetime.utcnow() + expires\_delta

27 to\_encode.update({"exp": expire})

28 return jwt.encode(to\_encode, SECRET\_KEY, algorithm="HS256")

29

30@app.post("/user/token")

31def login\_for\_access\_token(form\_data: OAuth2PasswordRequestForm = Depends()):

32 user = users\_db.get(form\_data.username)

33 if not user or user["password"] != form\_data.password:

34 raise HTTPException(status\_code=400, detail="Incorrect username or password")

35 access\_token = create\_access\_token(data={"sub": form\_data.username})

36 return {"access\_token": access\_token, "token\_type": "bearer"}

**3.3 Query Processing Microservice**

**Technologies Used:**

* FastAPI in Python
* JWT for token verification
* CORSMiddleware and custom middleware for OPTIONS handling

**Functionality:**

* **Authentication Verification:**  
  Uses OAuth2PasswordBearer to ensure that the incoming request includes a valid JWT token in its “Authorization” header.
* **Query Processing:**  
  Accepts POST requests to **/query/process\_query** with a JSON body containing the key **"query"**.
* **LLM Integration:**  
  Invokes a function **process\_query\_with\_llama** (in **llama3\_integration.py**) to generate an answer for the query.
* **Middleware for CORS:**  
  Both CORSMiddleware and a global middleware (or explicit OPTIONS route) are implemented to ensure proper handling of preflight requests.

**Key Code Snippet (Query Processing – main.py):**

Python

Collapse

1from fastapi import FastAPI, HTTPException, Depends, Request, Response

2from fastapi.middleware.cors import CORSMiddleware

3from fastapi.security import OAuth2PasswordBearer

4import jwt

5from llama3\_integration import process\_query\_with\_llama

6

7app = FastAPI(title="Query Processing Service")

8

9app.add\_middleware(

10 CORSMiddleware,

11 allow\_origins=["http://localhost:3000"],

12 allow\_credentials=True,

13 allow\_methods=["\*"],

14 allow\_headers=["\*"],

15)

16

17# Global middleware to automatically handle OPTIONS requests

18@app.middleware("http")

19async def options\_interceptor(request: Request, call\_next):

20 if request.method.upper() == "OPTIONS":

21 # Returning response with CORS headers by virtue of CORSMiddleware

22 return Response(status\_code=200)

23 return await call\_next(request)

24

25oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="/user/token")

26SECRET\_KEY = "your-secret-key" # Must match the secret from the user management service

27

28def verify\_token(token: str = Depends(oauth2\_scheme)):

29 try:

30 payload = jwt.decode(token, SECRET\_KEY, algorithms=["HS256"])

31 return payload

32 except jwt.PyJWTError:

33 raise HTTPException(status\_code=401, detail="Invalid token")

34

35@app.post("/query/process\_query")

36def process\_query(query: dict, token\_data: dict = Depends(verify\_token)):

37 received\_query = query.get("query", "")

38 print("Received query:", received\_query)

39 try:

40 answer = process\_query\_with\_llama(received\_query)

41 print("Generated answer:", answer)

42 return {"answer": answer}

43 except Exception as e:

44 print("Error in process\_query:", e)

45 raise HTTPException(status\_code=500, detail=f"Error processing query: {e}")

**3.4 Llama 3 Integration Module**

**Technologies Used:**

* Python, Hugging Face Transformers (for future integration)
* Currently, returns a dummy/simulated response for testing purposes

**Key Code Snippet (llama3\_integration.py):**

Python

1def process\_query\_with\_llama(query: str) -> str:

2 # For testing, we simulate the answer.

3 return f"Simulated answer for query: '{query}'"

**4. Deployment & Running the Project**

**4.1 Installation Prerequisites**

Ensure you have installed:

* Python 3.8 or newer
* Node.js and npm/yarn
* Required Python libraries (listed in the respective **requirements.txt** files)
* Git for version control (if needed)

**4.2 Running the Frontend**

1. Change directory to the **frontend** folder:

BASH

1cd frontend

1. Install dependencies:

BASH

1npm install

1. Start the React development server:

BASH

1npm start

The frontend will be available on **http://localhost:3000**.

**4.3 Running the Backend Services**

**User Management Service**

1. Navigate to the **backend/user\_management** folder:

BASH

1cd backend/user\_management

1. Install dependencies:

BASH

1pip install -r requirements.txt

1. Start the service:

BASH

1uvicorn main:app --reload --port 8000

**Query Processing Service**

1. Navigate to the **backend/query\_processing** folder:

BASH

1cd backend/query\_processing

1. Install dependencies:

BASH

1pip install -r requirements.txt

1. Start the service:

BASH

1uvicorn main:app --reload --port 8001

**5. Testing the System**

* **Using the Browser:**
  + Open **http://localhost:3000** in your browser.
  + Use the login screen with dummy credentials:  
    Email: **test@example.com**  
    Password: **test123**
  + Once logged in, navigate to the dashboard and enter a query (e.g., “tell about apple stocks”).
  + Verify that the query is processed and the answer is displayed.
* **Using Postman/Curl:**
  + Test login endpoint:

BASH

1curl -X POST "http://127.0.0.1:8000/user/token" \

2 -H "Content-Type: application/x-www-form-urlencoded" \

3 -d "username=test@example.com&password=test123"

* + Test query processing endpoint:

BASH

1curl -X POST "http://127.0.0.1:8001/query/process\_query" \

2 -H "Content-Type: application/json" \

3 -H "Authorization: Bearer <your\_token\_here>" \

4 -d '{"query": "tell about apple stocks"}'

**6. Additional Documentation & Future Enhancements**

* **Architecture Diagram:**  
  Refer to **docs/architecture.drawio.png** for a visual representation of the system.
* **Future Enhancements:**
  + Replace the dummy Llama integration with the actual Llama 3 model from Hugging Face Transformers.
  + Further modularize components and add persistent storage (e.g., PostgreSQL, MongoDB).
  + Introduce error logging and monitoring in the production environment.
  + Containerize the application using Docker.
* **Security Considerations:**
  + Use environment variables for sensitive configurations (SECRET\_KEY, API keys, etc.).
  + Harden Cross-Origin settings in production.
  + Use HTTPS and additional security headers on production deployments.

**7. Conclusion**

This detailed document outlines the design, implementation, and deployment of the Fintech LLM Chatbot. By integrating a modern front-end with robust FastAPI services and an LLM integration module, the system demonstrates a scalable solution for processing real-time financial queries. The architecture is built with future extensibility in mind, allowing for further enhancements such as real model integration, persistent storage, and containerized deployment.