System Design Document for Backend API Prototype

Project Overview

The goal of this project is to develop a backend API that supports user authentication, user management, and chat functionalities. The API is designed to be scalable, secure, and maintainable, utilizing a microservices architecture and adhering to design principles such as Atomic Design.

API URL

Base URL: `https://chatbot-ocs.azurewebsites.net`

1. Architecture Overview

1.1. System Architecture

The system follows a microservices architecture, with clear separation of concerns, reflecting the principles of Atomic Design:

- **API Gateway:** Serves as the single entry point for all client requests, managing routes and directing traffic to appropriate services.
- **Authentication Service:** Manages user login, logout, and token generation, ensuring secure access to the system.
- **User Service:** Handles user profile and friend-related functionalities, providing a modular approach to user management.
- **Chat Service:** Manages chat messaging functionalities, encapsulating chat-related operations within its own module.
- **Database:** MongoDB is used to store user and chat data, providing a flexible data structure that aligns with the overall design principles.

1.2. Technologies Used

- **Programming Language:** Python
- **Framework:** Azure Functions
- **Database:** MongoDB
- **Authentication:** JWT (JSON Web Tokens)

Libraries:

- **Flask:** For creating API routes and handling requests.
- **PyJWT:** For encoding and decoding JWT tokens.
- **bcrypt:** For hashing passwords.
- **pymongo:** For MongoDB database interactions.
- **logging:** For logging errors and information.

2. API Specifications

2.1. Authentication APIs

Login

- **Route:** `POST /api/auth/login`
- **Request Body:**

```
{
  "email": "user@example.com",
  "password": "string"
}
```

- **Response:**

```
{
  "message": "Login successful",
  "token": "jwt_token_string",
  "user": {
    "id": "user_id",
    "username": "string",
    "email": "user@example.com",
    "isEmailVerified": true,
    "friends": [],
    "chats": [],
    "online_status": "online",
    "last_login": "ISO 8601 date"
  }
}
```

Logout

- **Route:** `POST /api/auth/logout`
- **Headers:**
  - `Authorization: Bearer jwt_token_string`
- **Response:**

```
{
  "message": "Logout successful"
}
```

2.2. User Management APIs

Get User Profile

- **Route:** `GET /api/users/profile`
- **Headers:**
  - `Authorization: Bearer jwt_token_string`
- **Response:**

```
{
  "id": "user_id",
  "username": "string",
  "email": "user@example.com",
  "isEmailVerified": true,
  "friends": [],
  "chats": [],
  "online_status": "online",
  "last_login": "ISO 8601 date"
}
```

Update User Profile

- **Route:** `PUT /api/users/profile`
- **Headers:**
  - `Authorization: Bearer jwt_token_string`
- **Request Body:**

```
{
  "username": "new_username",
  "email": "new_email@example.com"
}
```

- **Response:**

```
{
  "message": "Profile updated successfully"
}
```

Add Friend

- **Route:** `POST /api/users/friends`
- **Headers:**
  - `Authorization: Bearer jwt_token_string`
- **Request Body:**

```
{
  "friendId": "friend_user_id"
}
```

- **Response:**

```
{
  "message": "Friend added successfully"
}
```

Remove Friend

- **Route:** `DELETE /api/users/friends`
- **Headers:**
  - `Authorization: Bearer jwt_token_string`
- **Request Body:**

```
{
  "friendId": "friend_user_id"
}
```

- **Response:**

```
{
  "message": "Friend removed successfully"
}
```

2.3. Chat APIs

Send Message

- **Route:** `POST /api/chat/send`
- **Headers:**
  - `Authorization: Bearer jwt_token_string`
- **Request Body:**

```
{
  "recipientId": "recipient_user_id",
  "message": "Hello!"
}
```

- **Response:**

```
{
  "message": "Message sent successfully"
}
```

Get Messages

- **Route:** `GET /api/chat/messages`
- **Headers:**
  - `Authorization: Bearer jwt_token_string`
- **Response:**

```
{
  "messages": [
    {
      "senderId": "sender_user_id",
      "recipientId": "recipient_user_id",
      "content": "Hello!",
      "timestamp": "ISO 8601 date"
    }
  ]
}
```

3. Setup and Running the Prototype

3.1. Prerequisites

- Python 3.x
- MongoDB Database
- Azure Account (for deploying Azure Functions)

3.2. Dependencies

Create a `requirements.txt` file with the following content:

```
Flask==2.0.1
PyJWT==2.0.0
bcrypt==3.2.0
pymongo==3.11.3
```

3.3. Setting Up the Environment

Clone the Repository

```
git clone https://github.com/Shweta-tiwari29/message
cd your-repo-directory
```

Install Dependencies

```
pip install -r requirements.txt
```

Set Up MongoDB

Create a MongoDB cluster on MongoDB Atlas or set up a local MongoDB instance. Get the connection string and update your code to connect to the database.

Configure Environment Variables

Create a `.env` file in the root directory with the following content:

```
JWT_SECRET=your_jwt_secret_key
MONGODB_URI=your_mongodb_connection_string
```

Run the Application

You can use Azure Functions Core Tools to run the functions locally:

```
func start
```

3.4. Testing the API

You can use tools like Postman or cURL to test the API endpoints. Ensure you include the JWT token in the Authorization header for protected routes.

4. Explanation of Libraries Used

- **Flask:** A lightweight WSGI web application framework that is easy to use for creating REST APIs.
- **PyJWT:** A Python library to work with JSON Web Tokens, enabling secure authentication and session management.
- **bcrypt:** A library for hashing passwords securely, which adds an additional layer of security to user authentication.
- **pymongo:** A Python driver for MongoDB, allowing for seamless interaction with the database.

5. Conclusion

This document provides a comprehensive overview of the backend API design, setup instructions, and dependencies used in the project. It serves as a guide for developers to understand and extend the system as needed.