Node.js Backend with Restify and WebSocket Integration for Chatbot Flow

Introduction:

The goal of this task is to develop a **Node.js backend** using **Restify** and **WebSockets** to manage a dynamic chatbot flow based on a JSON configuration. This backend will allow users to upload and download chatbot flow configurations, process conversations in real time, and integrate with OpenAI for intent detection. Additionally, it will maintain a history of user interactions.

The system should be designed with flexibility, scalability, and modularity in mind, ensuring smooth real-time communication and easy adaptability to different chatbot workflows.

Come up with a good JSON structure that can handle the given functional and technical requirements. You are free to use any technology that you find appropriate.

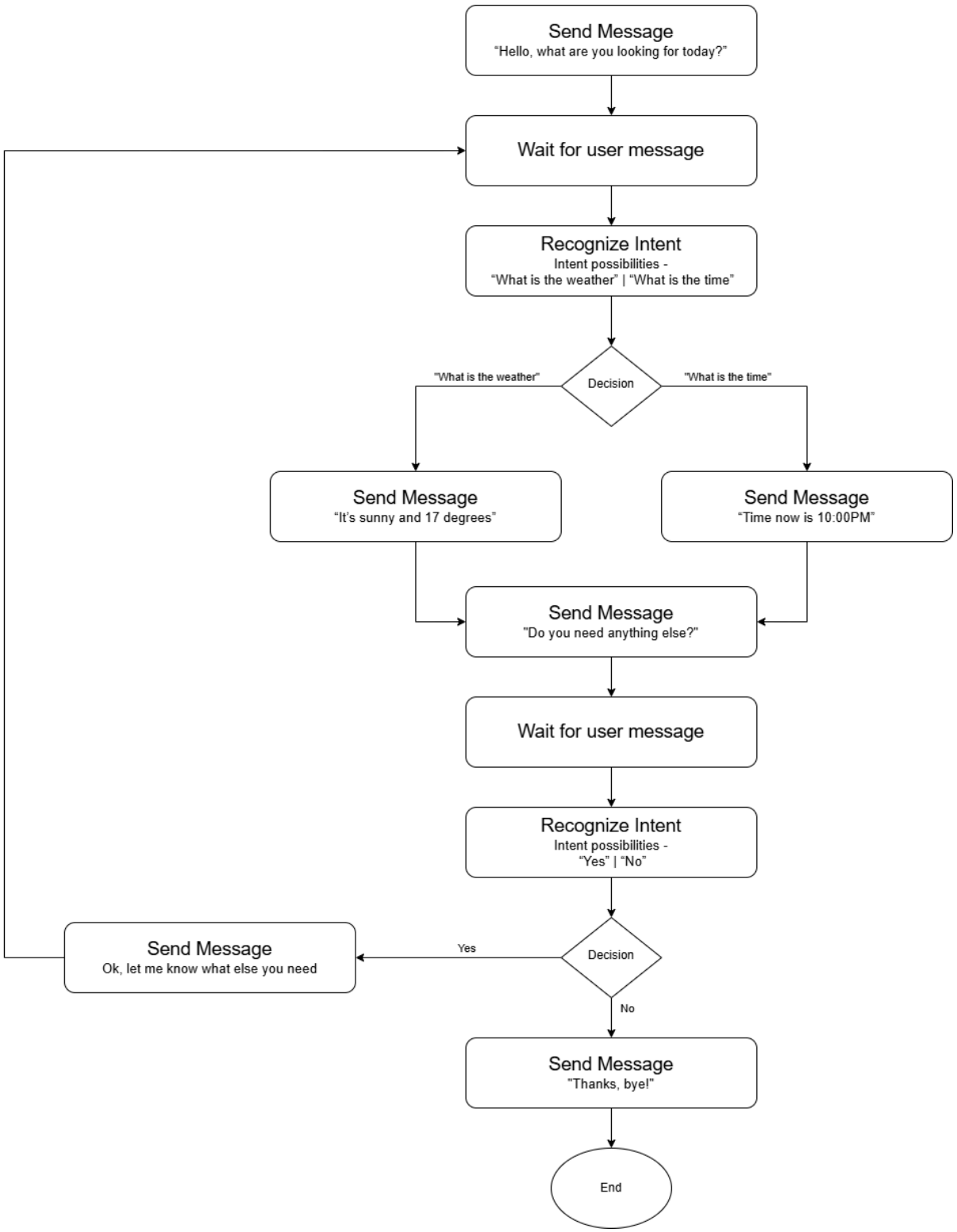
Functional Requirements:

1. **Endpoints:**
    - **Create JSON Config:**
        - Endpoint to upload a new or updated JSON configuration defining the chatbot flow.
        - It should accept the JSON object in the request body and store it.
        - Return success or failure status.
    - **Get JSON Config:**
        - Endpoint to get the current JSON configuration that defines the chatbot flow.
        - This should return the chatbot flow JSON as a response.
    - **WebSocket Endpoint:**
        - WebSocket server to facilitate real-time communication with the current chatbot JSON configuration.
        - This should allow a client (e.g., a frontend application) to send messages and interact with the chatbot flow.
        - The server should listen for incoming WebSocket messages, process the message based on the current JSON configuration, and send responses back in real-time.
2. **Chatbot Flow Blocks:**
    - The chatbot flow will be defined as blocks in the JSON. Each block represents a chatbot step with specific functionality.

- Supported blocks:
    - **Write a Message:** The chatbot sends a predefined message to the user.
    - **Wait for Response:** The chatbot waits for the user to respond.
    - **Detect Response Intent:** The chatbot uses OpenAI integration to analyze the user's response and detect the intent.
        1. The JSON must contain possible options as intents - e.g “Give me the weather”, “Give me travel offers”, “Give me restaurant recommendations”. The OpenAI integration must decide which one matches best, given the response from the user.
        2. Based on the detected intent or response, the chatbot can decide which block to proceed to (this can be another block defined in the JSON, even blocks that have already passed).
        3. Handle exceptions, like cases where the LLM cannot understand the user’s message

3. **Conversation History:**
    - Store the history of all conversations, including the sent messages, bot responses, and flow steps taken.
4. **Real-time WebSocket Communication:**
    - The conversation should be started when the user connects to the WebSocket endpoint.
    - Clients can send messages via WebSocket and receive responses based on the defined blocks.
    - The server should handle incoming WebSocket messages, process them through the flow defined in the JSON, and respond accordingly.

Example Flow:



Technical Requirements:

Self-Contained Application:

- The application should be fully self-contained, requiring no additional setup or configuration to run.

RESTful API Design:

- Develop the service endpoints following **REST principles** using **Restify** in a **Node.js** backend.

Build & Dependency Management:

- Use appropriate tools like **npm** or **yarn** for managing dependencies and building the project.

Use of Design Patterns:

- Implement design patterns where appropriate to enhance **code quality, maintainability, and modularity.**

Code Structure:

- Follow a **clear separation of concerns**, organizing the code into e.g. **controllers, services, and utilities** for better maintainability.

Unit Testing:

- Include a couple of **unit tests** to validate the main chatbot flow logic.

API Documentation:

- Provide a readme file containing a **complete and accurate documentation** for all API endpoints, including request and response examples, and startup instructions.

Docker:

- **Containerize** the application with **Docker** to ensure consistent behavior across different environments.

Proper Use of a Git Repository:

- Maintain the code in a **Git repository** with a **clear commit history**, following best practices for version control, including feature branches.

Database or storage:

- Use any means of storage as you find appropriate for this project.

Time for completion: 5-7 days.