**Chat Application**

**I have built a real-time group chat application using Spring Boot and WebSocket. It allows users to create or join chat rooms using a unique room ID. All messages are stored in MongoDB, so when a user joins a room, they can see the previous messages. WebSocket ensures instant message delivery to all users in the room. The system is scalable, supports multiple chat rooms, and provides a smooth chatting experience with minimal latency.**

**Group Chat Application Using Spring Boot and WebSocket**

**Objective**

The goal is to develop a real-time group chat application using **Spring Boot** and **WebSocket** with **MongoDB** for message persistence.

**Functional Requirements**

**1. Room Management**

**(a) Create Room**

* Users can create a new chat room by entering a unique name or ID.
* The system validates that the room ID is unique.
* The new room details are saved in MongoDB.

**(b) Join Room**

* Users can join an existing chat room using a room ID.
* If the room exists, users can view past messages and start chatting.
* If the room does not exist, an error message is displayed.

**2. Messaging**

* Users can send and receive messages in real time.
* Messages are stored in **MongoDB** under their respective room.

**3. Real-Time Communication**

* WebSocket is used to ensure instant message delivery to all users in the room.

**Non-Functional Requirements**

**1. Scalability**

* The system should support multiple users and rooms at the same time.
* **MongoDB** should handle chat data efficiently.

**2. Performance**

* Messages should be delivered with minimal latency.
* Quick retrieval of chat history from **MongoDB** when a user joins a room.

**3. User-Friendly Interface**

* Clear options for **creating** and **joining** rooms.
* A simple chat UI with room management features.

**System Design**

**1. Database Schema (MongoDB)**

**(a) Room Collection**

{

"\_id": "ObjectId",

"roomId": "string",

"messages": [

{

"sender": "string",

"message": "string",

"timestamp": "Date"

}

]

}

**Fields:**

* **id**: Auto-generated unique identifier for the room.
* **roomId**: Unique string identifier for the room.
* **messages**: An array of message objects, each containing:
  + **sender**: Name of the user.
  + **message**: Text content of the message.
  + **timestamp**: Time when the message was sent.

**Technology Stack**

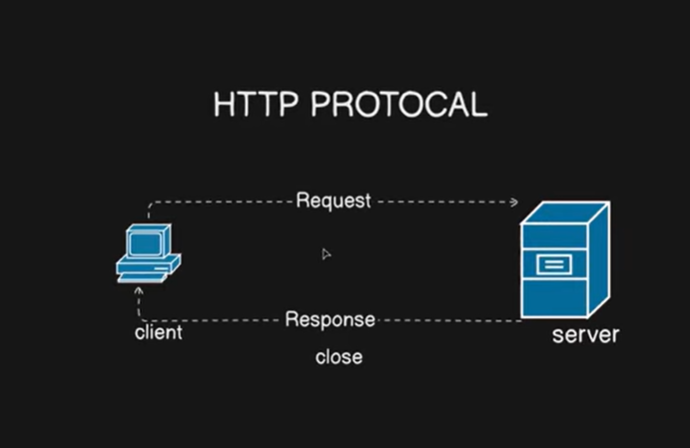
* **Backend:** Spring Boot, WebSocket, MongoDB
* **Frontend (optional):** React/Angular for UI
* **Database:** MongoDB for storing rooms and messages

detailed notes on **WebSockets vs REST API, STOMP, and SockJS**:

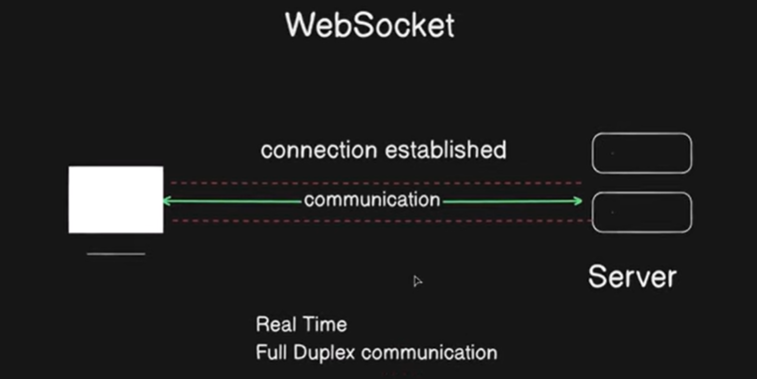
**Why Use WebSockets Instead of REST API?**

**Issues with REST API**

1. **Request-Response Model**
   * In a REST API, the client sends a request to the server, and the server responds.
   * This process is repeated every time a new message or update is needed.
2. **Polling and Refreshing**
   * To receive real-time updates, the client needs to keep sending requests (polling) or refresh the page.
   * This increases network traffic and server load.
3. **Connection Overhead**
   * Each REST request creates a new connection, processes the request, sends a response, and then closes the connection.
   * This is inefficient for applications that require real-time communication, such as chat applications, live stock updates, or gaming.



**WebSocket Protocol**



**What is WebSocket?**

* WebSockets solve the above issues by **creating a persistent, duplex connection** between the client and server.
* Unlike REST, WebSockets **do not close the connection** after each message, making real-time communication seamless.

**How WebSockets Work?**

1. The client **establishes a connection** with the server using an HTTP handshake.
2. Once the connection is established, it remains **open** until either the client or server closes it.
3. Both **client and server can send messages** to each other in real time without delays.

**Advantages of WebSockets Over REST**

✅ **Real-time communication** – No need to refresh the page or send repeated requests.  
✅ **Low latency** – Messages are sent instantly without the delay of creating a new request.  
✅ **Efficient use of resources** – A single connection remains open, reducing overhead.

**STOMP Protocol**

**Why Use STOMP?**

* **WebSockets are low-level**, meaning developers need to handle many complexities like message formats, routing, and handling disconnections.
* **STOMP (Simple Text-Oriented Messaging Protocol)** is a protocol that runs over WebSockets, making it easier to manage messaging.

**How STOMP Works?**

1. The client subscribes to a particular channel (topic).
2. When the server has new data, it **pushes messages** to all subscribed clients.
3. This simplifies real-time communication by removing the need for manual message handling.

**Benefits of STOMP**

✅ **Easier to implement** – Provides a structured way to handle messages over WebSockets.  
✅ **Supports multiple messaging patterns** – Pub/Sub (Publish/Subscribe) and Queue-based communication.  
✅ **Standardized message format** – Uses a simple text-based protocol.

**SockJS – Fallback Mechanism for WebSockets**

**Why Use SockJS?**

* **WebSockets are not supported on all browsers or networks** (e.g., firewalls might block them).
* **SockJS is a JavaScript library** that provides a **fallback mechanism** to ensure communication even if WebSockets are unavailable.

**How SockJS Works?**

* It first tries to establish a WebSocket connection.
* If WebSockets are not available, it falls back to other techniques like:
  + **AJAX Long Polling** – Client sends a request and waits for the server to respond.
  + **Streaming** – Server continuously sends updates without closing the connection.

**Benefits of SockJS**

✅ **Ensures compatibility across all browsers and networks.**  
✅ **Automatic fallback to alternative transport mechanisms.**  
✅ **Helps maintain real-time communication when WebSockets fail.**