**1️⃣ Definition — What is “Spring Boot + AI”?**

In simple terms:

**Spring Boot + AI** means building backend services using the **Spring Boot framework** that integrate **Artificial Intelligence (AI) capabilities** — either by calling external AI APIs (like OpenAI GPT-4, Azure OpenAI, Google Gemini) or running **local AI models** (like LLaMA, Mistral, Falcon) inside your infrastructure.

It’s not just “calling AI from Java” — it’s about combining **Spring Boot’s enterprise features**:

* REST / WebSocket APIs
* Security
* Rate limiting
* Distributed architecture  
  … with **AI processing pipelines** like:
* Chatbots
* Summarization
* Search with RAG (Retrieval-Augmented Generation)
* Predictive analytics

**2️⃣ Why Use Spring Boot for AI?**

Spring Boot gives:

* **Rapid development** → Auto-configured REST endpoints for AI integration.
* **Enterprise features** → Security, caching, resilience, distributed processing.
* **Scalability** → Works in microservices with Kubernetes, Docker, etc.
* **Flexibility** → Easy to swap AI backends (local → cloud API).

AI gives:

* **Natural language understanding** → Chatbots, search, automation.
* **Content generation** → Text, code, images.
* **Decision making** → Predictions, recommendations.

**3️⃣ Core Definitions in This Context**

| **Term** | **Deep Dive Definition** |
| --- | --- |
| **LLM (Large Language Model)** | An AI model trained on massive datasets to understand and generate human-like text. Examples: GPT-4, LLaMA-3, Mistral. |
| **RAG (Retrieval-Augmented Generation)** | A method where you **retrieve** relevant knowledge from a database/vector store and feed it into the AI prompt so the model can answer with context. |
| **Embedding** | A numerical vector representation of text, enabling similarity search in vector databases. |
| **Vector Database** | A special DB (like Pinecone, Milvus, Qdrant) that stores embeddings for semantic search. |
| **Prompt Engineering** | Crafting precise instructions for the AI to get desired outputs. |
| **Token** | Smallest unit of text AI models process; affects cost and speed in API-based models. |
| **Inference** | Running a trained AI model on new inputs to generate predictions/outputs. |
| **Streaming Response** | Sending AI responses token-by-token to the client, improving perceived latency. |

**4️⃣ Integration Patterns for Spring Boot + AI**

**A. Direct API Calls (Hosted AI)**

* Use AI APIs like OpenAI, Anthropic, Azure AI.
* Spring Boot calls these via REST using RestTemplate or WebClient.
* ✅ Pros: Easy to implement, high accuracy.
* ❌ Cons: Costs money per request, data privacy concerns.

**B. Local AI Models**

* Run LLaMA, GPT4All, Mistral locally via **Ollama**, **LangChain4j**, or **Deep Java Library (DJL)**.
* ✅ Pros: Private, no API costs.
* ❌ Cons: Needs GPU/CPU resources, harder scaling.

**C. Hybrid (Local + Cloud)**

* Use local models for routine queries, cloud AI for complex reasoning.
* Good for **cost + performance** balance.

**5️⃣ Example Spring Boot AI Architecture**

java

CopyEdit

Client (Web / Mobile / Chat UI)

│

▼

Spring Boot API Layer

├── Security (JWT, OAuth2)

├── Rate Limiting (Bucket4j + Redis)

├── AI Service Layer

│ ├── Hosted AI API (OpenAI, Azure)

│ ├── Local Model (Ollama, GPT4All)

│ └── RAG Pipeline (LangChain4j + Vector DB)

├── WebSocket/SSE Streaming

└── Persistence (Postgres, Redis, Vector DB)

**6️⃣ Deep Dive — RAG Flow in Spring Boot**

1. **User Query** → Comes via REST or WebSocket.
2. **Security Layer** → Validates token & user permissions.
3. **Rate Limit Check** → Throttles if needed.
4. **Embedding Generation** → Convert query into a vector using an embedding model.
5. **Vector Search** → Query vector DB for relevant chunks.
6. **Prompt Assembly** → Combine retrieved text + user query.
7. **LLM Inference** → Send prompt to local/cloud LLM.
8. **Streaming Response** → Send tokens back in real-time.

**7️⃣ Key Libraries & Tools**

| **Purpose** | **Tool** |
| --- | --- |
| LLM Access | **LangChain4j**, **Ollama**, **DJL** |
| RAG | **LangChain4j**, **Milvus**, **Weaviate**, **Pinecone** |
| Embeddings | OpenAI API, **SentenceTransformers** via DJL |
| Security | Spring Security, OAuth2, JWT |
| Rate Limiting | Bucket4j, Resilience4j |
| Streaming | Spring WebFlux, WebSocket, Server-Sent Events |
| Infra | Docker, Kubernetes |

**8️⃣ Real-World Use Cases**

* **Customer Support Chatbot** → Answers questions from internal KB.
* **Document Search & Summarization** → Find & summarize PDFs.
* **Fraud Detection** → AI-enhanced anomaly detection pipeline.
* **Code Assistant** → Integrate AI to help developers inside an enterprise tool.
* **Multilingual Support** → AI translation + Spring Boot REST APIs.

**9️⃣ Key Challenges**

1. **Latency** → Large models can be slow; streaming helps.
2. **Cost Control** → Use local inference or rate limits.
3. **Security & Privacy** → Don’t leak sensitive data in prompts.
4. **Model Selection** → Balance between accuracy and performance.
5. **Scalability** → Use async processing, caching, and load balancing.

**1️⃣ 📚 Key Definitions for Spring Boot + AI**

| **Term** | **Definition** |
| --- | --- |
| **Spring Boot** | A Java framework that simplifies creating standalone, production-ready applications with minimal configuration. |
| **AI (Artificial Intelligence)** | Technology enabling machines to mimic human intelligence tasks like understanding, reasoning, and generating content. |
| **LLM (Large Language Model)** | AI model trained on large datasets to understand and generate text (e.g., GPT-4, LLaMA). |
| **RAG (Retrieval-Augmented Generation)** | Combines vector search with AI: retrieves relevant documents and passes them to the model for accurate answers. |
| **Embedding** | A vector representation of text used for semantic search in vector databases. |
| **Vector Database** | Specialized DB (e.g., Milvus, Weaviate, Pinecone) for storing and querying embeddings. |
| **Prompt Engineering** | Designing prompts to get desired results from AI models. |
| **Streaming Response** | Sending AI output to clients in real-time, token by token. |
| **Inference** | Running a trained AI model to generate predictions/answers for new inputs. |
| **Token** | The smallest unit AI processes (like a sub-word). Affects API cost & performance. |

**2️⃣ 🔄 Architecture Diagrams**

**A. High-Level Spring Boot + AI Flow**

mermaid

CopyEdit

flowchart TD

User[User Query] --> API[Spring Boot REST/WebSocket API]

API --> Sec[Security (JWT/OAuth2)]

Sec --> Rate[Rate Limiting (Bucket4j)]

Rate --> RAG[Retrieve Context (Vector DB)]

RAG --> Prompt[Build Prompt with Context]

Prompt --> LLM[AI Model (Local/Cloud)]

LLM --> Stream[Stream Response to Client]

**B. RAG Inside Spring Boot**

mermaid

CopyEdit

sequenceDiagram

participant U as User

participant API as Spring Boot API

participant EMB as Embedding Model

participant VDB as Vector DB

participant LLM as LLaMA / GPT API

U->>API: Send question

API->>EMB: Generate embedding vector

API->>VDB: Search for similar docs

VDB-->>API: Return top matches

API->>LLM: Send prompt + retrieved docs

LLM-->>API: Return generated answer

API-->>U: Stream answer back

**3️⃣ 🛠 Code Examples**

**AI Service using LangChain4j + Ollama**

java

CopyEdit

@Service

public class AiChatService {

private final OllamaChatModel model;

public AiChatService(@Value("${ollama.url}") String url,

@Value("${ollama.model}") String modelName) {

this.model = OllamaChatModel.builder()

.baseUrl(url)

.modelName(modelName)

.build();

}

public String chat(String prompt) {

return model.generate(prompt);

}

}

**Rate Limiting with Bucket4j + Redis**

java

CopyEdit

@Component

public class RedisRateLimiterFilter extends OncePerRequestFilter {

private final RedissonClient redissonClient;

public RedisRateLimiterFilter(RedissonClient redissonClient) {

this.redissonClient = redissonClient;

}

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,

FilterChain filterChain) throws IOException, ServletException {

RMapCache<String, GridBucketState> cache = redissonClient.getMapCache("rate-limits");

ProxyManager<String> manager = Bucket4j.extension(Redisson.class).proxyManagerForMap(cache);

Bucket bucket = manager.builder()

.build(request.getRemoteAddr(), () ->

Bucket4j.configurationBuilder()

.addLimit(Bandwidth.classic(10, Refill.intervally(10, Duration.ofMinutes(1))))

.build()

);

if (bucket.tryConsume(1)) {

filterChain.doFilter(request, response);

} else {

response.setStatus(429);

response.getWriter().write("Rate limit exceeded");

}

}

}

**WebSocket Chat Endpoint**

java

CopyEdit

@Controller

public class ChatWebSocket {

private final AiChatService aiChatService;

public ChatWebSocket(AiChatService aiChatService) {

this.aiChatService = aiChatService;

}

@MessageMapping("/chat")

@SendTo("/topic/replies")

public String handleMessage(String message) {

return aiChatService.chat(message);

}

}

**4️⃣ 📊 Pros & Cons Table for AI Integration Approaches**

| **Approach** | **Pros** | **Cons** | **Use Cases** |
| --- | --- | --- | --- |
| **Cloud AI API (OpenAI, Azure)** | High accuracy, minimal setup, latest models | Cost per request, data privacy concerns | Chatbots, content generation |
| **Local AI Model (Ollama, GPT4All)** | Private, no API cost, offline | Needs server resources, may be slower | On-prem compliance, internal tools |
| **Hybrid (Local + Cloud)** | Cost control, privacy for sensitive data | Complexity in routing | Enterprise AI with mixed workloads |
| **RAG (Vector DB + AI)** | Accurate, domain-specific answers | Needs extra infra (vector DB) | Document search, enterprise knowledge base |

**1️⃣ AI Model Integration Options**

You have two main choices: **Local AI Models** or **Hosted AI APIs**.

**A. Local AI Models**

*(Runs on your own server or on-premises — good for privacy, no external API costs)*

* [**GPT4All Java bindings**](https://github.com/nomic-ai/gpt4all)
  + Supports running local LLaMA, MPT, Falcon models.
  + Integrate via Java native bindings (JNI) or through a local REST API layer.
* [**LangChain4j**](https://github.com/langchain4j/langchain4j)
  + Java-native LangChain library.
  + Supports local LLaMA via GPT4All, Ollama, and remote APIs.
  + Great for chaining prompts, retrieval-augmented generation (RAG).
* **Ollama**
  + Simple CLI & server for LLaMA, Mistral, Gemma, etc.
  + Call via HTTP from Spring Boot.
* **Deep Java Library (DJL)**
  + Directly run PyTorch, TensorFlow, ONNX models in Java.
  + Has LLaMA and HuggingFace integrations.

**B. Hosted AI APIs**

*(Managed cloud — less setup, faster scaling)*

* **OpenAI API** (for GPT-4, GPT-4o-mini)
* **Azure OpenAI** (enterprise-grade with Azure security & compliance)
* **Anthropic Claude API**
* **Cohere API**
* **Google Gemini API**

📌 **Spring Boot Tip:** Wrap the model calls in a @Service so switching from local to hosted is easy.

**2️⃣ Security for AI APIs**

* **Spring Security** → OAuth2, JWT, or API Key-based auth.
* **API Gateway** (Kong, Spring Cloud Gateway, or Nginx) for:
  + Authentication & Authorization
  + Centralized API key management
* **mTLS (Mutual TLS)** for high-trust internal AI API calls.
* **Input Sanitization** → Avoid prompt injection & malicious payloads.
* **Output Filtering** → Check AI-generated text for compliance before sending to clients.

**3️⃣ Rate Limiting**

* **Bucket4j**
  + Java in-memory/distributed token bucket rate limiting.
  + Works with Redis, Hazelcast, or JCache.
* **Resilience4j**
  + Also has rate limiting, retry, bulkhead patterns.
* **Spring Cloud Gateway RateLimiter**
  + Redis-based, works well for distributed environments.
* **Kong API Gateway / Envoy**
  + If you need infra-level rate limiting before hitting Spring Boot.

**4️⃣ Chatbot + RAG Architecture**

* **LangChain4j** (Java) or **LangChain (Python)** with REST bridge
* **Vector Databases** for retrieval:
  + **Milvus**, **Weaviate**, **Pinecone**, **Qdrant**
* **Embedding Models**:
  + OpenAI embeddings API
  + Local: all-MiniLM-L6-v2 (via SentenceTransformers + DJL)
* **WebSocket / SSE in Spring Boot** for live-streaming AI responses
* **Spring WebFlux** if you need reactive streaming to the frontend.

**5️⃣ Example Spring Boot AI Stack**

**Local LLaMA Chatbot with Rate Limiting & Security**

pgsql

CopyEdit

Spring Boot

├─ Spring Security (JWT Auth)

├─ Bucket4j (Rate Limit)

├─ LangChain4j (Prompt + RAG pipeline)

├─ Ollama / GPT4All (Local AI)

├─ Milvus (Vector DB for context)

├─ Spring WebFlux (Streaming responses)

✅ **Quick Library List for Integration**

| **Purpose** | **Library/Framework** |
| --- | --- |
| Local AI Model | GPT4All Java Bindings, Ollama, DJL |
| LLM Chaining / RAG | LangChain4j |
| Hosted AI Calls | OpenAI Java SDK, Azure AI SDK |
| Vector Storage | Milvus, Weaviate, Pinecone |
| Rate Limiting | Bucket4j, Resilience4j |
| Security | Spring Security, OAuth2, JWT |
| Streaming | Spring WebFlux, Server-Sent Events |
| Gateway | Spring Cloud Gateway, Kong |

If you want, I can **draw a complete Spring Boot + AI architecture diagram** showing how LLaMA/GPT4All, rate limiting, and security fit together.  
Do you want me to make that diagram next?

**You said:**

spring bood code implenntation ?

**ChatGPT said:**

Got it ✅ — you want a **Spring Boot implementation example** that integrates AI (LLaMA via GPT4All/Ollama), adds **security (JWT)**, and **rate limiting (Bucket4j)**, and exposes a **chatbot API**.

I’ll give you a **minimal but production-ready** code skeleton you can build on.

**1️⃣ Maven Dependencies (pom.xml)**

xml

CopyEdit

<dependencies>

<!-- Spring Boot Web -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- Spring Security for JWT -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<!-- LangChain4j (supports GPT4All, Ollama, OpenAI) -->

<dependency>

<groupId>dev.langchain4j</groupId>

<artifactId>langchain4j</artifactId>

<version>0.27.0</version>

</dependency>

<!-- Bucket4j Rate Limiting -->

<dependency>

<groupId>com.github.vladimir-bukhtoyarov</groupId>

<artifactId>bucket4j-core</artifactId>

<version>8.8.0</version>

</dependency>

<!-- JWT -->

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.5</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

</dependencies>

**2️⃣ application.properties**

properties

CopyEdit

server.port=8080

# Ollama local server

ollama.api.url=http://localhost:11434/api/generate

ollama.model=llama2

jwt.secret=my-secret-key

**3️⃣ AI Service Layer**

Example with **Ollama API** (you can swap to GPT4All or OpenAI easily).

java

CopyEdit

@Service

public class AiService {

@Value("${ollama.api.url}")

private String ollamaApiUrl;

@Value("${ollama.model}")

private String ollamaModel;

private final RestTemplate restTemplate = new RestTemplate();

public String chat(String userPrompt) {

Map<String, Object> request = Map.of(

"model", ollamaModel,

"prompt", userPrompt,

"stream", false

);

ResponseEntity<Map> response = restTemplate.postForEntity(ollamaApiUrl, request, Map.class);

return response.getBody().get("response").toString();

}

}

**4️⃣ Rate Limiting Filter (Bucket4j)**

java

CopyEdit

@Component

public class RateLimitFilter extends OncePerRequestFilter {

private final Bucket bucket;

public RateLimitFilter() {

Bandwidth limit = Bandwidth.classic(5, Refill.greedy(5, Duration.ofMinutes(1)));

this.bucket = Bucket4j.builder().addLimit(limit).build();

}

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,

FilterChain filterChain) throws ServletException, IOException {

if (bucket.tryConsume(1)) {

filterChain.doFilter(request, response);

} else {

response.setStatus(429);

response.getWriter().write("Rate limit exceeded. Try later.");

}

}

}

**5️⃣ JWT Security Config**

java

CopyEdit

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

return http.csrf().disable()

.authorizeHttpRequests(auth -> auth

.requestMatchers("/auth/\*\*").permitAll()

.anyRequest().authenticated()

)

.httpBasic(Customizer.withDefaults())

.build();

}

}

**6️⃣ Chat Controller**

java

CopyEdit

@RestController

@RequestMapping("/api/chat")

public class ChatController {

private final AiService aiService;

public ChatController(AiService aiService) {

this.aiService = aiService;

}

@PostMapping

public ResponseEntity<String> ask(@RequestBody Map<String, String> payload) {

String prompt = payload.get("prompt");

return ResponseEntity.ok(aiService.chat(prompt));

}

}

**7️⃣ Example Call**

bash

CopyEdit

curl -X POST http://localhost:8080/api/chat \

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

-d '{"prompt":"Explain Spring Boot to a beginner"}'

**8️⃣ How This Works**

* **Security**: You can enhance JWT parsing & authentication.
* **Rate Limiting**: 5 requests/min per instance.
* **AI Service**: Calls Ollama (local LLaMA model) — can be replaced with GPT4All or OpenAI.
* **Controller**: Simple REST endpoint for chatbot requests.

If you want, I can now make a **full working Spring Boot + AI chatbot repo** with:

* ✅ JWT Auth
* ✅ Bucket4j distributed (Redis)
* ✅ LangChain4j with LLaMA RAG example
* ✅ WebSocket streaming responses

That would give you a production-grade starting point.  
Do you want me to build that next?

**You said:**

✅ JWT Auth ✅ Bucket4j distributed (Redis) ✅ LangChain4j with LLaMA RAG example ✅ WebSocket streaming responses

**ChatGPT said:**

Perfect 👍 — that’s basically a **production-grade Spring Boot AI chatbot stack**.

Here’s how we’ll structure it:

**1️⃣ Project Structure**

bash

CopyEdit

springboot-ai-chat/

├── pom.xml

├── src/main/java/com/example/ai/

│ ├── config/ # Security, Redis, LangChain4j configs

│ ├── controller/ # REST + WebSocket endpoints

│ ├── service/ # AI service, RAG, embeddings

│ ├── security/ # JWT utils & filters

│ └── model/ # DTOs

└── src/main/resources/

└── application.properties

**2️⃣ Maven Dependencies**

xml

CopyEdit

<dependencies>

<!-- Spring Boot Web + WebSocket -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-websocket</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- Spring Security -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<!-- JWT -->

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.5</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

<!-- Bucket4j + Redis -->

<dependency>

<groupId>com.github.vladimir-bukhtoyarov</groupId>

<artifactId>bucket4j-redis</artifactId>

<version>8.8.0</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

<!-- LangChain4j (Java LLM + RAG) -->

<dependency>

<groupId>dev.langchain4j</groupId>

<artifactId>langchain4j</artifactId>

<version>0.27.0</version>

</dependency>

<dependency>

<groupId>dev.langchain4j</groupId>

<artifactId>langchain4j-ollama</artifactId>

<version>0.27.0</version>

</dependency>

</dependencies>

**3️⃣ application.properties**

properties

CopyEdit

server.port=8080

# JWT

jwt.secret=my-secret-key

jwt.expiration=3600000

# Redis

spring.redis.host=localhost

spring.redis.port=6379

# Ollama

ollama.url=http://localhost:11434

ollama.model=llama2

# Vector DB (in-memory for now)

vector.store.dir=./vectors

**4️⃣ JWT Utility**

java

CopyEdit

@Component

public class JwtUtil {

@Value("${jwt.secret}")

private String secret;

@Value("${jwt.expiration}")

private long expiration;

public String generateToken(String username) {

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date(System.currentTimeMillis() + expiration))

.signWith(Keys.hmacShaKeyFor(secret.getBytes()), SignatureAlgorithm.HS256)

.compact();

}

public String validateToken(String token) {

return Jwts.parserBuilder()

.setSigningKey(Keys.hmacShaKeyFor(secret.getBytes()))

.build()

.parseClaimsJws(token)

.getBody()

.getSubject();

}

}

**5️⃣ Distributed Rate Limiter (Bucket4j + Redis)**

java

CopyEdit

@Component

public class RedisRateLimiterFilter extends OncePerRequestFilter {

private final RedissonClient redissonClient;

private final String RATE\_LIMIT\_KEY = "ai-api-rate";

public RedisRateLimiterFilter(RedissonClient redissonClient) {

this.redissonClient = redissonClient;

}

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,

FilterChain filterChain) throws IOException, ServletException {

RMapCache<String, GridBucketState> buckets = redissonClient.getMapCache("rate-limits");

ProxyManager<String> bucketsManager = Bucket4j.extension(Redisson.class).proxyManagerForMap(buckets);

Bucket bucket = bucketsManager.builder()

.build(RATE\_LIMIT\_KEY, () ->

Bucket4j.configurationBuilder()

.addLimit(Bandwidth.classic(10, Refill.greedy(10, Duration.ofMinutes(1))))

.build()

);

if (bucket.tryConsume(1)) {

filterChain.doFilter(request, response);

} else {

response.setStatus(429);

response.getWriter().write("Rate limit exceeded. Try again later.");

}

}

}

**6️⃣ LangChain4j LLaMA RAG Service**

java

CopyEdit

@Service

public class AiChatService {

private final OllamaChatModel model;

public AiChatService(@Value("${ollama.url}") String url,

@Value("${ollama.model}") String modelName) {

this.model = OllamaChatModel.builder()

.baseUrl(url)

.modelName(modelName)

.build();

}

public String chat(String userMessage) {

return model.generate(userMessage);

}

}

**7️⃣ WebSocket Streaming Controller**

java

CopyEdit

@Controller

public class ChatWebSocket {

private final AiChatService aiChatService;

public ChatWebSocket(AiChatService aiChatService) {

this.aiChatService = aiChatService;

}

@MessageMapping("/chat")

@SendTo("/topic/replies")

public String handleMessage(String message) {

return aiChatService.chat(message);

}

}

**Frontend (JS) Example**:

javascript

CopyEdit

const socket = new SockJS('/ws');

const stompClient = Stomp.over(socket);

stompClient.connect({}, () => {

stompClient.subscribe('/topic/replies', (msg) => {

console.log("AI says:", msg.body);

});

stompClient.send("/app/chat", {}, "Hello AI!");

});