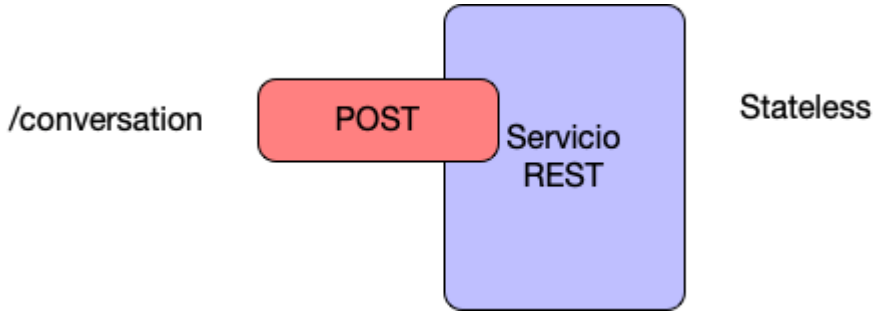
Mantener estado entre peticiones es uno de los retos más habituales al construir aplicaciones conversacionales con modelos de lenguaje. En una API REST, cada petición HTTP es independiente, pero el usuario espera que el asistente recuerde lo que se dijo antes. Con Spring AI no es necesario inventar una solución con `HashMap`, variables estáticas o sesiones improvisadas: el framework ya ofrece mecanismos nativos para gestionar memoria conversacional de forma ordenada y extensible.



Mantener contexto conversacional con Spring AI

Cuando integramos un modelo de lenguaje en una aplicación Spring Boot, normalmente enviamos un prompt y recibimos una respuesta. El problema aparece cuando el usuario continúa la conversación: "¿puedes resumir lo anterior?" o "usa el mismo formato que antes". Si cada petición se procesa de forma aislada, el modelo no tiene forma de saber a qué se refiere. Por eso necesitamos asociar cada mensaje a una conversación concreta.

Spring AI resuelve este escenario mediante el concepto de `ChatMemory`. Esta abstracción permite guardar y recuperar mensajes previos de una conversación para incluirlos automáticamente en nuevas llamadas al modelo. La idea es sencilla: cada conversación tiene un identificador, por ejemplo `conversationId`, y Spring AI usa ese identificador para saber qué historial debe recuperar antes de invocar al modelo.

Lo importante es que no hablamos de mantener estado manualmente en un controlador ni de crear un `Map<String, List>` en memoria. Esa solución puede servir para una prueba rápida, pero se vuelve frágil en cuanto hay reinicios, múltiples instancias o necesidad de limpiar mensajes antiguos. Con `ChatMemory`, Spring AI proporciona una forma nativa, más

limpia y preparada para evolucionar hacia almacenamiento persistente.

Usar ChatMemory nativo sin mapas en memoria

Una forma habitual de usar esta capacidad es configurar un `ChatClient` con un advisor de memoria. El advisor se encarga de interceptar la llamada, recuperar los mensajes anteriores asociados a la conversación y añadirlos al prompt enviado al modelo. Después, también registra la nueva interacción en la memoria. Todo esto ocurre sin que el controlador tenga que manipular directamente listas de mensajes.

Un ejemplo básico en Spring Boot podría ser el siguiente:

```
@Configuration
public class AiConfig {

    @Bean
    ChatMemory chatMemory() {
        return MessageWindowChatMemory.builder()
                .maxMessages(20)
                .build();
    }

    @Bean
    ChatClient chatClient(ChatClient.Builder builder, ChatMemory
chatMemory) {
        return builder
                .defaultAdvisors(
MessageChatMemoryAdvisor.builder(chatMemory).build()
                )
                .build();
    }
```

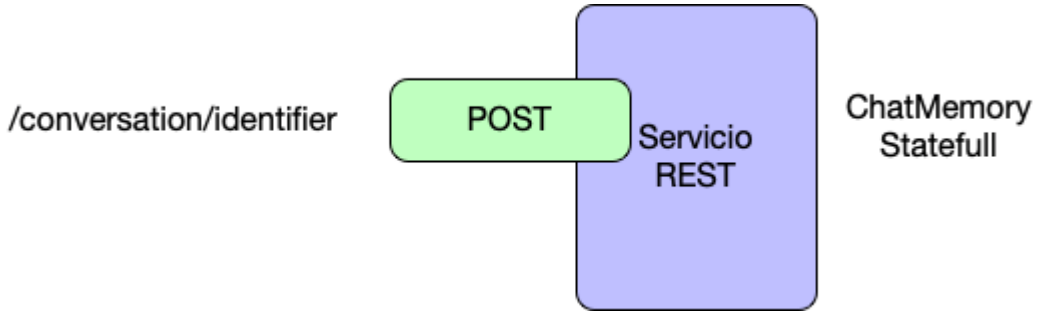
```
}
```

Y el controlador podría recibir un identificador de conversación en cada petición:

```
@RestController
@RequestMapping("/api/chat")
public class ChatController {

    private final ChatClient chatClient;

    public ChatController(ChatClient chatClient) {
        this.chatClient = chatClient;
    }

    @PostMapping("/{conversationId}")
    public String chat(
            @PathVariable String conversationId,
            @RequestBody String message
    ) {
        return chatClient.prompt()
                .user(message)
                .advisors(advisor -> advisor.param(
                        ChatMemory.CONVERSATION_ID,
                        conversationId
                ))
                .call()
                .content();
    }
}
```

Con este enfoque, si el cliente llama varias veces a `/api/chat/usuario-123` o `/api/chat/conversacion-abc`, Spring AI asociará esas peticiones al mismo historial.

`MessageWindowChatMemory` permite además limitar cuántos mensajes se conservan, evitando enviar conversaciones demasiado largas al modelo. En algunas versiones de Spring AI, el nombre de la constante puede variar y encontrarse en clases relacionadas con los advisors de memoria, pero el concepto es el mismo: pasar un identificador de conversación al advisor.



Persistir conversaciones en base de datos

La memoria en proceso es útil para desarrollo, demostraciones o aplicaciones simples, pero no siempre es suficiente. Si la aplicación se reinicia, el historial se pierde. Si se despliega en Kubernetes con varias réplicas, cada instancia puede tener una memoria diferente. Y si se necesita auditar conversaciones o retomarlas días después, hace falta una capa persistente.

Spring AI contempla este escenario separando la abstracción de memoria del lugar donde se almacena. En lugar de acoplar la lógica conversacional a una estructura manual, se puede usar una implementación respaldada por base de datos o crear una implementación propia de repositorio de memoria. La aplicación seguiría usando `ChatMemory` y el `ChatClient` de forma similar, pero los mensajes quedarían guardados en un sistema externo.

A nivel conceptual, la tabla de conversaciones suele incluir un identificador de conversación, el rol del mensaje, el contenido, la fecha y quizá metadatos como usuario, canal o tenant. No hace falta entrar en una arquitectura compleja desde el inicio, pero sí conviene diseñar pensando en producción: limpieza de historiales antiguos, límites de tokens, cumplimiento de privacidad y soporte para varias instancias de la aplicación compartiendo el mismo almacenamiento.

Mantener estado entre peticiones con Spring AI no tiene por qué resolverse con soluciones caseras. Usando `ChatMemory` y los advisors del `ChatClient`, podemos conservar contexto conversacional de forma nativa, limpia y extensible. Para entornos simples, una memoria de ventana puede ser suficiente; para producción, lo recomendable es evolucionar hacia persistencia en base de datos manteniendo la misma idea central: cada petición lleva un `conversationId` y Spring AI se encarga de reconstruir el contexto adecuado para el modelo.

- ¿Que es Spring AI ?¿Para que sirve?
- Curso Spring AI : Inteligencia artificial en Java
- Utilizando jQuery Native Selectors
- ¿Que es un UUID?
- Java List to Map y el uso de Collectors