Chat Memory

Chat Memory

Large language models (LLMs) are stateless, meaning they do not retain information about previous interactions. This can be a limitation when you want to maintain context or state across multiple interactions. To address this, Spring AI provides chat memory features that allow you to store and retrieve information across multiple interactions with the LLM.

The `ChatMemory` abstraction allows you to implement various types of memory to support different use cases. The underlying storage of the messages is handled by the `ChatMemoryRepository`, whose sole responsibility is to store and retrieve messages. It's up to the `ChatMemory` implementation to decide which messages to keep and when to remove them. Examples of strategies could include keeping the last N messages, keeping messages for a certain time period, or keeping messages up to a certain token limit.

Before choosing a memory type, it's essential to understand the difference between chat memory and chat history.

- **Chat Memory**. The information that a large-language model retains and uses to maintain contextual awareness throughout a conversation.
- **Chat History**. The entire conversation history, including all messages exchanged between the user and the model.

The `ChatMemory` abstraction is designed to manage the *chat memory*. It allows you to store and retrieve messages that are relevant to the current conversation context. However, it is not the best fit for storing the *chat history*. If you need to maintain a complete record of all the messages exchanged, you should consider using a different approach, such as relying on Spring Data for efficient storage and retrieval of the complete chat history.

Quick Start

Spring AI auto-configures a `ChatMemory` bean that you can use directly in your application. By default, it uses an in-memory repository to store messages (`InMemoryChatMemoryRepository`) and a `MessageWindowChatMemory` implementation to manage the conversation history. If a different repository is already configured (e.g., Cassandra, JDBC, or Neo4j), Spring AI will use that instead.

```java
@Autowired
ChatMemory chatMemory;
```

The following sections will describe further the different memory types and repositories available in Spring AI.

Memory Types

The `ChatMemory` abstraction allows you to implement various types of memory to suit different use cases. The choice of memory type can significantly impact the performance and behavior of your application. This section describes the built-in memory types provided by Spring AI and their characteristics.

Message Window Chat Memory

`MessageWindowChatMemory` maintains a window of messages up to a specified maximum size. When the number of messages exceeds the maximum, older messages are removed while preserving system messages. The default window size is 20 messages.

```java
MessageWindowChatMemory memory = MessageWindowChatMemory.builder()
    .maxMessages(10)
    .build();
```

This is the default message type used by Spring AI to auto-configure a `ChatMemory` bean.

Memory Storage

Spring AI offers the `ChatMemoryRepository` abstraction for storing chat memory. This section describes the built-in repositories provided by Spring AI and how to use them, but you can also implement your own repository if needed.

In-Memory Repository

`InMemoryChatMemoryRepository` stores messages in memory using a `ConcurrentHashMap`.

By default, if no other repository is already configured, Spring AI auto-configures a `ChatMemoryRepository` bean of type `InMemoryChatMemoryRepository` that you can use directly in your application.

```java
@Autowired
ChatMemoryRepository chatMemoryRepository;
```

If you'd rather create the `InMemoryChatMemoryRepository` manually, you can do so as follows:

```java
ChatMemoryRepository repository = new InMemoryChatMemoryRepository();
```

JdbcChatMemoryRepository

`JdbcChatMemoryRepository` is a built-in implementation that uses JDBC to store messages in a relational database. It supports multiple databases out-of-the-box and is suitable for applications that require persistent storage of chat memory.

First, add the following dependency to your project:

Maven | Gradle

```xml
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-starter-model-chat-memory-repository-jdbc</artifactId>
</dependency>
```

Spring AI provides auto-configuration for the `JdbcChatMemoryRepository`, that you can use directly in your application.

```java
@Autowired
JdbcChatMemoryRepository chatMemoryRepository;

ChatMemory chatMemory = MessageWindowChatMemory.builder()
    .chatMemoryRepository(chatMemoryRepository)
```

```
    .maxMessages(10)
    .build();
```

If you'd rather create the `JdbcChatMemoryRepository` manually, you can do so by providing a `JdbcTemplate` instance and a `JdbcChatMemoryRepositoryDialect`:

JAVA

```
ChatMemoryRepository chatMemoryRepository = JdbcChatMemoryRepository.builder()
    .jdbcTemplate(jdbcTemplate)
    .dialect(new PostgresChatMemoryDialect())
    .build();

ChatMemory chatMemory = MessageWindowChatMemory.builder()
    .chatMemoryRepository(chatMemoryRepository)
    .maxMessages(10)
    .build();
```

Supported Databases and Dialect Abstraction

Spring AI supports multiple relational databases via a dialect abstraction. The following databases are supported out-of-the-box:

- PostgreSQL
- MySQL / MariaDB
- SQL Server
- HSQLDB

The correct dialect can be auto-detected from the JDBC URL when using `JdbcChatMemoryRepositoryDialect.from(DataSource)`. You can extend support for other databases by implementing the `JdbcChatMemoryRepositoryDialect` interface.

Configuration Properties

Property	Description	Default Value
`spring.ai.chat.memory.repository.jdbc.initialize-schema`	Controls when to initialize the schema. Values: `embedded` (default), `always`, `never`.	`embedded`

`spring.ai.chat.memory.repository.jdbc.schema`	Location of the schema script to use for initialization. Supports `classpath:` URLs and platform placeholders.	`classpath:org/springframework/@@platform@@.sql`
`spring.ai.chat.memory.repository.jdbc.platform`	Platform to use in initialization scripts if the @@platform@@ placeholder is used.	*auto-detected*

Schema Initialization

The auto-configuration will automatically create the `SPRING_AI_CHAT_MEMORY` table on startup, using a vendor-specific SQL script for your database. By default, schema initialization runs only for embedded databases (H2, HSQL, Derby, etc.).

You can control schema initialization using the `spring.ai.chat.memory.repository.jdbc.initialize-schema` property:

PROPERTIES
```
spring.ai.chat.memory.repository.jdbc.initialize-schema=embedded # Only for embedded DBs (default)
spring.ai.chat.memory.repository.jdbc.initialize-schema=always   # Always initialize
spring.ai.chat.memory.repository.jdbc.initialize-schema=never    # Never initialize (useful with Flyway/Liquibase)
```

To override the schema script location, use:

PROPERTIES
```
spring.ai.chat.memory.repository.jdbc.schema=classpath:/custom/path/schema-mysql.sql
```

Extending Dialects

To add support for a new database, implement the `JdbcChatMemoryRepositoryDialect` interface and provide SQL for selecting, inserting, and deleting messages. You can then pass your custom dialect to the repository builder.

JAVA
```
ChatMemoryRepository chatMemoryRepository = JdbcChatMemoryRepository.builder()
    .jdbcTemplate(jdbcTemplate)
    .dialect(new MyCustomDbDialect())
    .build();
```

CassandraChatMemoryRepository

`CassandraChatMemoryRepository` uses Apache Cassandra to store messages. It is suitable for applications that require persistent storage of chat memory, especially for availability, durability, scale, and when taking advantage of time-to-live (TTL) feature.

`CassandraChatMemoryRepository` has a time-series schema, keeping record of all past chat windows, valuable for governance and auditing. Setting time-to-live to some value, for example three years, is recommended.

To use `CassandraChatMemoryRepository` first, add the dependency to your project:

Maven | **Gradle**

XML

```xml
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-starter-model-chat-memory-repository-cassandra</artifactId>
</dependency>
```

Spring AI provides auto-configuration for the `CassandraChatMemoryRepository` that you can use directly in your application.

JAVA

```java
@Autowired
CassandraChatMemoryRepository chatMemoryRepository;

ChatMemory chatMemory = MessageWindowChatMemory.builder()
    .chatMemoryRepository(chatMemoryRepository)
    .maxMessages(10)
    .build();
```

If you'd rather create the `CassandraChatMemoryRepository` manually, you can do so by providing a `CassandraChatMemoryRepositoryConfig` instance:

JAVA

```java
ChatMemoryRepository chatMemoryRepository = CassandraChatMemoryRepository
    .create(CassandraChatMemoryConfig.builder().withCqlSession(cqlSession));

ChatMemory chatMemory = MessageWindowChatMemory.builder()
    .chatMemoryRepository(chatMemoryRepository)
    .maxMessages(10)
    .build();
```

Configuration Properties

Property	Description	Default Value
`spring.cassandra.contactPoints`	Host(s) to initiate cluster discovery	`127.0.0.1`
`spring.cassandra.port`	Cassandra native protocol port to connect to	`9042`
`spring.cassandra.localDatacenter`	Cassandra datacenter to connect to	`datacenter1`
`spring.ai.chat.memory.cassandra.time-to-live`	Time to live (TTL) for messages written in Cassandra	
`spring.ai.chat.memory.cassandra.keyspace`	Cassandra keyspace	`springframework`
`spring.ai.chat.memory.cassandra.messages-column`	Cassandra column name for messages	`springframework`
`spring.ai.chat.memory.cassandra.table`	Cassandra table	`ai_chat_memory`
`spring.ai.chat.memory.cassandra.initialize-schema`	Whether to initialize the schema on startup.	`true`

Schema Initialization

The auto-configuration will automatically create the `ai_chat_memory` table.

You can disable the schema initialization by setting the property `spring.ai.chat.memory.repository.cassandra.initialize-schema` to `false`.

Neo4j ChatMemoryRepository

`Neo4jChatMemoryRepository` is a built-in implementation that uses Neo4j to store chat messages as nodes and relationships in a property graph database. It is suitable for applications that want to leverage Neo4j's graph capabilities for chat memory persistence.

First, add the following dependency to your project:

Maven | Gradle

```xml
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-starter-model-chat-memory-repository-neo4j</artifactId>
</dependency>
```

Spring AI provides auto-configuration for the `Neo4jChatMemoryRepository`, which you can use directly in your application.

JAVA

```java
@Autowired
Neo4jChatMemoryRepository chatMemoryRepository;

ChatMemory chatMemory = MessageWindowChatMemory.builder()
    .chatMemoryRepository(chatMemoryRepository)
    .maxMessages(10)
    .build();
```

If you'd rather create the `Neo4jChatMemoryRepository` manually, you can do so by providing a Neo4j `Driver` instance:

JAVA

```java
ChatMemoryRepository chatMemoryRepository = Neo4jChatMemoryRepository.builder()
    .driver(driver)
    .build();

ChatMemory chatMemory = MessageWindowChatMemory.builder()
    .chatMemoryRepository(chatMemoryRepository)
    .maxMessages(10)
    .build();
```

Configuration Properties

Property	Description	Default Value
`spring.ai.chat.memory.repository.neo4j.sessionLabel`	The label for the nodes that store conversation sessions	`Session`
`spring.ai.chat.memory.repository.neo4j.messageLabel`	The label for the nodes that store messages	`Message`
`spring.ai.chat.memory.repository.neo4j.toolCallLabel`	The label for nodes that store tool calls (e.g. in Assistant Messages)	`ToolCall`
`spring.ai.chat.memory.repository.neo4j.metadataLabel`	The label for nodes that store message metadata	`Metadata`
`spring.ai.chat.memory.repository.neo4j.toolResponseLabel`	The label for the nodes that store tool responses	`ToolResponse`

`spring.ai.chat.memory.repository.neo4j.mediaLabel`	The label for the nodes that store media associated with a message	`Media`

Index Initialization

The Neo4j repository will automatically ensure that indexes are created for conversation IDs and message indices to optimize performance. If you use custom labels, indexes will be created for those labels as well. No schema initialization is required, but you should ensure your Neo4j instance is accessible to your application.

Memory in Chat Client

When using the ChatClient API, you can provide a `ChatMemory` implementation to maintain conversation context across multiple interactions.

Spring AI provides a few built-in Advisors that you can use to configure the memory behavior of the `ChatClient`, based on your needs.

WARNING

Currently, the intermediate messages exchanged with a large-language model when performing tool calls are not stored in the memory. This is a limitation of the current implementation and will be addressed in future releases. If you need to store these messages, refer to the instructions for the User Controlled Tool Execution.

- `MessageChatMemoryAdvisor`. This advisor manages the conversation memory using the provided `ChatMemory` implementation. On each interaction, it retrieves the conversation history from the memory and includes it in the prompt as a collection of messages.
- `PromptChatMemoryAdvisor`. This advisor manages the conversation memory using the provided `ChatMemory` implementation. On each interaction, it retrieves the conversation history from the memory and appends it to the system prompt as plain text.
- `VectorStoreChatMemoryAdvisor`. This advisor manages the conversation memory using the provided `VectorStore` implementation. On each interaction, it retrieves the conversation history from the vector store and appends it to the system message as plain text.

For example, if you want to use `MessageWindowChatMemory` with the `MessageChatMemoryAdvisor`, you can configure it as follows:

JAVA

```
ChatMemory chatMemory = MessageWindowChatMemory.builder().build();

ChatClient chatClient = ChatClient.builder(chatModel)
```

```
    .defaultAdvisors(MessageChatMemoryAdvisor.builder(chatMemory).build())
    .build();
```

When performing a call to the `ChatClient`, the memory will be automatically managed by the `MessageChatMemoryAdvisor`. The conversation history will be retrieved from the memory based on the specified conversation ID:

JAVA

```
String conversationId = "007";

chatClient.prompt()
    .user("Do I have license to code?")
    .advisors(a -> a.param(ChatMemory.CONVERSATION_ID, conversationId))
    .call()
    .content();
```

PromptChatMemoryAdvisor

Custom Template

The `PromptChatMemoryAdvisor` uses a default template to augment the system message with the retrieved conversation memory. You can customize this behavior by providing your own `PromptTemplate` object via the `.promptTemplate()` builder method.

NOTE

The `PromptTemplate` provided here customizes how the advisor merges retrieved memory with the system message. This is distinct from configuring a `TemplateRenderer` on the `ChatClient` itself (using `.templateRenderer()`), which affects the rendering of the initial user/system prompt content **before** the advisor runs. See ChatClient Prompt Templates for more details on client-level template rendering.

The custom `PromptTemplate` can use any `TemplateRenderer` implementation (by default, it uses `StPromptTemplate` based on the StringTemplate engine). The important requirement is that the template must contain the following two placeholders:

- an `instructions` placeholder to receive the original system message.
- a `memory` placeholder to receive the retrieved conversation memory.

VectorStoreChatMemoryAdvisor

Custom Template

The `VectorStoreChatMemoryAdvisor` uses a default template to augment the system message with the retrieved conversation memory. You can customize this behavior by providing your own `PromptTemplate` object via the `.promptTemplate()` builder method.

NOTE

The `PromptTemplate` provided here customizes how the advisor merges retrieved memory with the system message. This is distinct from configuring a `TemplateRenderer` on the `ChatClient` itself (using `.templateRenderer()`), which affects the rendering of the initial user/system prompt content **before** the advisor runs. See ChatClient Prompt Templates for more details on client-level template rendering.

The custom `PromptTemplate` can use any `TemplateRenderer` implementation (by default, it uses `StPromptTemplate` based on the StringTemplate engine). The important requirement is that the template must contain the following two placeholders:

- an `instructions` placeholder to receive the original system message.
- a `long_term_memory` placeholder to receive the retrieved conversation memory.

Memory in Chat Model

If you're working directly with a `ChatModel` instead of a `ChatClient`, you can manage the memory explicitly:

JAVA

```
// Create a memory instance
ChatMemory chatMemory = MessageWindowChatMemory.builder().build();
String conversationId = "007";

// First interaction
UserMessage userMessage1 = new UserMessage("My name is James Bond");
chatMemory.add(conversationId, userMessage1);
ChatResponse response1 = chatModel.call(new Prompt(chatMemory.get(conversationId)));
chatMemory.add(conversationId, response1.getResult().getOutput());

// Second interaction
UserMessage userMessage2 = new UserMessage("What is my name?");
chatMemory.add(conversationId, userMessage2);
ChatResponse response2 = chatModel.call(new Prompt(chatMemory.get(conversationId)));
chatMemory.add(conversationId, response2.getResult().getOutput());

// The response will contain "James Bond"
```