Docs

Designing AI services

An *AI Service* employs a declarative approach to define interactions with the LLM, serving as the pivotal interaction point between your application and the LLM. It operates as an intermediary, known as an *ambassador*.

Purpose

The *AI Service* serves as the core connection point between your application and the LLM. It abstracts the LLM specifics, encapsulating and declaring all interactions within a singular interface.

Leveraging @RegisterAiService

The `@RegisterAiService` annotation is pivotal for registering an *AI Service*, placed on an interface:

```
@RegisterAiService
public interface MyAiService {
    // methods.
}
```

Once registered, you can inject the *AI Service* into your application:

```
@Inject MyAiService service;
```

IMPORTANT

The beans created by `@RegisterAiService` are `@RequestScoped` by default. The reason for this is that it enables removing chat Configuring the Context (Memory) objects. This is a good default when a service is used during when handling an HTTP request, but it's inappropriate in CLIs or in WebSockets (WebSocket support is expected to improve in the near future). For example when using a service in a CLI, it makes sense to have the service be `@ApplicationScoped` and the extension allows this simply if the service is annotated with `@ApplicationScoped` .

AI method declaration

Within the interface annotated with `@RegisterAiService` , you model interactions with the LLM

using *methods*. These methods accept parameters and are annotated with `@SystemMessage` and `@UserMessage` to define instructions directed to the LLM:

```
@SystemMessage("You are a professional poet.")
@UserMessage("""
    Write a poem about {topic}. The poem should be {lines} lines long. Then send
this poem by email.
""")
String writeAPoem(String topic, int lines);
```

System Message

The `@SystemMessage` annotation defines the scope and initial instructions, serving as the first message sent to the LLM. It delineates the AI service's role in the interaction:

```
@SystemMessage("""
    You are working for a bank, processing reviews about financial products. Triage
reviews into positive and negative ones, responding with a JSON document.
    """
)
```

User Message (Prompt)

The `@UserMessage` annotation defines primary instructions dispatched to the LLM. It typically encompasses requests and the expected response format:

```
@UserMessage("""
    Your task is to process the review delimited by ---.
    Apply a sentiment analysis to the review to determine if it is positive or
negative, considering various languages.

    For example:
    - "I love your bank, you are the best!" is a 'POSITIVE' review
    - "J'adore votre banque" is a 'POSITIVE' review
    - "I hate your bank, you are the worst!" is a 'NEGATIVE' review

    Respond with a JSON document containing:
    - the 'evaluation' key set to 'POSITIVE' if the review is positive, 'NEGATIVE'
otherwise
    - the 'message' key set to a message thanking or apologizing to the customer.
These messages must be polite and match the review's language.

    ---
    {review}
    ---
```

```
""")
TriagedReview triage(String review);
```

Parameter Passing and Referencing

AI methods can take parameters referenced in system and user messages using the `{parameter}` syntax:

```
@SystemMessage("You are a professional poet")
@UserMessage("""
    Write a poem about {topic}. The poem should be {lines} lines long. Then send
this poem by email.
""")
String writeAPoem(String topic, int lines);
```

AI Method Return Type

If the *prompt* defines the JSON response format precisely, you can map the response directly to an object:

```
// ... See above for the prompt
TriagedReview triage(String review);
```

In this instance, Quarkus automatically creates an instance of `TriagedReview` from the LLM's JSON response.

Receiving User Message as a Parameter

For situations requiring the user message to be passed as a parameter, you can use the `@UserMessage` annotation on a parameter. Exercise caution with this feature, especially when the AI has access to *tools*:

```
String chat(@UserMessage String userMessage);
```

The annotated parameter should be of type `String`.

Receiving MemoryId as a Parameter

The *memory* encompasses the cumulative context of the interaction with the LLM. To manage statelessness in LLMs, the complete context must be exchanged between the LLM and the AI service.

Hence, the AI Service can store the latest messages in a *memory*, often termed *context*. The

`@MemoryId` annotation enables referencing the memory for a specific user in the AI method:

```
String chat(@MemoryId int memoryId, @UserMessage String userMessage);
```

We'll explore an alternative approach to avoid manual memory handling in the Configuring the Context (Memory) section.

Configuring the Chat Language Model

While LLMs are the base AI models, the chat language model builds upon them, enabling chat-like interactions. If you have a single chat language model, no specific configuration is required.

However, when multiple model providers are present in the application (such as OpenAI, Azure OpenAI, HuggingFace, etc.) each model needs to be given a name, which is then referenced by the AI service like so:

```
@RegisterAiService(modelName="m1")
```

The configuration of the various models could look like so:

```
# ensure that the model with the name 'm1', is provided by OpenAI
quarkus.langchain4j.m1.chat-model.provider=openai
# ensure that the model with the name 'm2', is provided by HuggingFace
quarkus.langchain4j.m2.chat-model.provider=huggingface

# configure the various aspects of each model
quarkus.langchain4j.openai.m1.api-key=sk-...
quarkus.langchain4j.huggingface.m2.api-key=sk-...
```

Configuring the Context (Memory)

As LLMs are stateless, the memory — comprising the interaction context — must be exchanged each time. To prevent storing excessive messages, it's crucial to evict older messages.

The `chatMemoryProviderSupplier` attribute of the `@RegisterAiService` annotation enables configuring the `dev.langchain4j.memory.chat.ChatMemoryProvider`. The default value of this annotation is `RegisterAiService.BeanChatMemoryProviderSupplier.class` which means that the `AiService` will use whatever `ChatMemoryProvider` bean is configured by the application or the default one provided by the extension.

The extension provides a default implementation of `ChatMemoryProvider` which does two things:

- It uses whatever bean `dev.langchain4j.store.memory.chat.ChatMemoryStore` bean is configured, as the backing store. The default implementation is `dev.langchain4j.store.memory.chat.InMemoryChatMemoryStore`
  - If the application provides its own `ChatMemoryStore` bean, that will be used instead of the default `InMemoryChatMemoryStore`,
- It leverages the available configuration options under `quarkus.langchain4j.chat-memory` to construct the `ChatMemoryProvider`.
  - The default configuration values result in the usage of `dev.langchain4j.memory.chat.MessageWindowChatMemory` with a window size of ten.
  - By setting `quarkus.langchain4j.chat-memory.type=token-window`, a `dev.langchain4j.memory.chat.TokenWindowChatMemory` will be used. Note that this requires the presence of a `dev.langchain4j.model.Tokenizer` bean.

IMPORTANT

The topic of `ChatMemory` cleanup is of paramount importance in order to avoid having the application terminate with `out of memory` errors. For this reason, the extension automatically removes all the `ChatMemory` objects from the underlying `ChatMemoryStore` when the AI Service goes out of scope (recall from our discussion about [scope] that such bean are `@RequestScoped` be default).

However, in cases where more fine-grained control is needed (which is the case when the bean is declared as `@Singleton` or `@ApplicationScoped`) then `io.quarkiverse.langchain4j.ChatMemoryRemover` should be used to manually remove elements.

CAUTION

When using an AiService that is expected to use to chat memory, it is very important to use `@MemoryId` (as mentioned in a later section). Failure to do so, can lead to unexpected and hard to debug results.

If your use case requires that no memory should be used, then be sure to use `@RegisterAiService(chatMemoryProviderSupplier = RegisterAiService.NoChatMemoryProviderSupplier.class)`

Advanced usage

Although the extension's default `ChatMemoryProvider` is very configurable making unnecessary in most cases to resort to a custom implementation, such a capability is possible. Here is a possible example:

```
package io.quarkiverse.langchain4j.samples;
```

```
import jakarta.inject.Singleton;

import dev.langchain4j.memory.ChatMemory;
import dev.langchain4j.memory.chat.ChatMemoryProvider;
import dev.langchain4j.store.memory.chat.ChatMemoryStore;

@Singleton
public class CustomChatMemoryProvider implements ChatMemoryProvider {

    private final ChatMemoryStore store;

    public CustomChatMemoryProvider() {
        this.store = createCustomStore();
    }

    private static ChatMemoryStore createCustomStore() {
        // TODO: provide some kind of custom store
        return null;
    }

    @Override
    public ChatMemory get(Object memoryId) {
        return createCustomMemory(memoryId);
    }

    private static ChatMemory createCustomMemory(Object memoryId) {
        // TODO: implement using memoryId and store
        return null;
    }
}
```

If for some reason different AI services need to have a different `ChatMemoryProvider` (i.e. not use the globally available bean), this is possible by configuring the `chatMemoryProviderSupplier` attribute of the `@RegisterAiService` annotation and implementing as custom provider. Here is a possible example:

```
package io.quarkiverse.langchain4j.samples;

import java.util.function.Supplier;

import dev.langchain4j.memory.ChatMemory;
import dev.langchain4j.memory.chat.ChatMemoryProvider;
```

```
import dev.langchain4j.memory.chat.MessageWindowChatMemory;
import dev.langchain4j.store.memory.chat.InMemoryChatMemoryStore;

public class CustomProvider implements Supplier<ChatMemoryProvider> {

    private final InMemoryChatMemoryStore store = new InMemoryChatMemoryStore();

    @Override
    public ChatMemoryProvider get() {
        return new ChatMemoryProvider() {

            @Override
            public ChatMemory get(Object memoryId) {
                return MessageWindowChatMemory.builder()
                        .maxMessages(20)
                        .id(memoryId)
                        .chatMemoryStore(store)
                        .build();
            }
        };
    }
}
```

and configuring the AiService as so:

```
@RegisterAiService(
    chatMemoryProviderSupplier = MySmallMemoryProvider.class)
```

TIP

For non-memory-reliant LLM interactions, you may skip memory configuration.

@MemoryId

In cases involving multiple users, ensure each user has a unique memory ID and pass this ID to the AI method:

```
String chat(@MemoryId int memoryId, @UserMessage String userMessage);
```

Also, remember to clear out users to prevent memory issues.

Configuring Tools

Tools are methods that LLMs can invoke to access additional data. These methods, declared using

the `@Tool` annotation, should be part of a bean:

```
@ApplicationScoped
public class CustomerRepository implements PanacheRepository<Customer> {

    @Tool("get the customer name for the given customerId")
    public String getCustomerName(long id) {
        return find("id", id).firstResult().name;
    }

}
```

The `@Tool` annotation can provide a description of the action, aiding the LLM in tool selection. The `@RegisterAiService` annotation allows configuring the tool provider:

```
@RegisterAiService(tools = {TransactionRepository.class, CustomerRepository.class
})
```

IMPORTANT

Ensure you configure the memory provider when using tools.

IMPORTANT

Be cautious to avoid exposing destructive operations via tools.

More information about tools is available in the Agent and Tools page.

Configuring a Document Retriever

A document retriever fetches data from an external source and provides it to the LLM. It helps by sending only the relevant data, considering the LLM's context limitations.

This guidance aims to cover all crucial aspects of designing AI services with Quarkus, ensuring robust and efficient interactions with LLMs.

Moderation

By default, @RegisterAiService annotated interfaces don't moderate content. However, users can opt in to having the LLM moderate content by annotating the method with `@Moderate` .

For moderation to work, the following criteria need to be met:

- A CDI bean for `dev.langchain4j.model.moderation.ModerationModel` must be config-

ured (the `quarkus-langchain4j-openai` and `quarkus-langchain4j-azure-openai` provide one out of the box)

- The interface must be configured with `@RegisterAiService(moderationModelSupplier = RegisterAiService.BeanModerationModelSupplier.class)`

Advanced usage

An alternative to providing a CDI bean is to configure the interface with `@RegisterAiService(moderationModelSupplier = MyCustomSupplier.class)` and implement `MyCustomModerationSupplier` like so:

```
import dev.langchain4j.model.moderation.ModerationModel;

public class MyCustomModerationSupplier implements Supplier<ModerationModel> {

    @Override
    public ModerationModel get(){
        // TODO: implement
    }

}
```

Observability

Observability is built into services created via `@RegisterAiService` and is provided in the following form:

- Metrics are enabled when `quarkus-micrometer` is part of the application
- Traces are enabled when `quarkus-opentelemetry` is part of the application

Metrics

Each AI method is automatically timed and the timer data is available using the `langchain4j.aiservices.$interface_name.$method_name` template for the name.

For example, if the AI service looks like:

```
@RegisterAiService
public interface PoemAiService {

    @SystemMessage("You are a professional poet")
    @UserMessage("Write a poem about {topic}. The poem should be {lines} lines
long")
```

```
    String writeAPoem(String topic, int lines);
}
```

and one chooses to use `quarkus-micrometer-registry-prometheus`, then the metrics could be:

```
# TYPE langchain4j_aiservices counter
# HELP langchain4j_aiservices
langchain4j_aiservices_total{aiservice="MyAiService",exception="none",method="write
APoem",result="success"} 5.0

# TYPE langchain4j_aiservices_seconds_max gauge
# HELP langchain4j_aiservices_seconds_max
langchain4j_aiservices_seconds_max{aiservice="MyAiService",method="writeAPoem"}
7.725769221
# TYPE langchain4j_aiservices_seconds summary
# HELP langchain4j_aiservices_seconds
langchain4j_aiservices_seconds_count{aiservice="MyAiService",method="writeAPoem"}
5.0
langchain4j_aiservices_seconds_sum{aiservice="MyAiService",method="writeAPoem"}
30.229575906
```

Tracing

Each AI method creates its own span using the `langchain4j.aiservices.$interface_name.$method_name` template for the name. Furthermore, tool invocations also create a span using `langchain4j.tools.$tool_name` template for the name.

For example, if the AI service looks like:

```
@RegisterAiService(tools = EmailService.class)
public interface PoemAiService {

    @SystemMessage("You are a professional poet")
    @UserMessage("Write a poem about {topic}. The poem should be {lines} lines
long. Then send this poem by email.")
    String writeAPoem(String topic, int lines);

}
```

a tool that looks like:

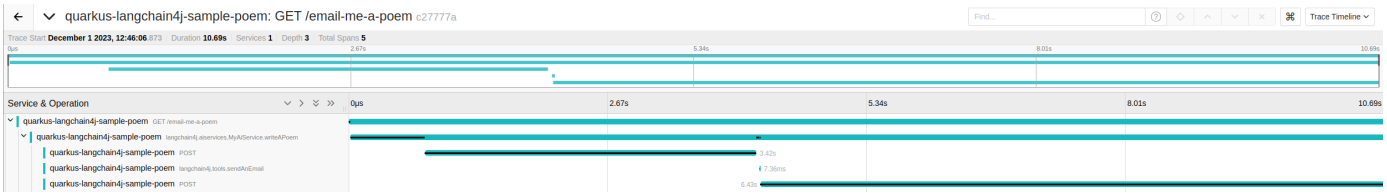
```
@ApplicationScoped
public class EmailService {
```

```
    @Inject
    Mailer mailer;

    @Tool("send the given content by email")
    public void sendAnEmail(String content) {
        Log.info("Sending an email: " + content);
        mailer.send(Mail.withText("sendMeALetter@quarkus.io", "A poem for you",
content));
    }

}
```

and invocation of the AI service that looks like:

```
@Path("/email-me-a-poem")
public class EmailMeAPoemResource {

    private final MyAiService service;

    public EmailMeAPoemResource(MyAiService service) {
        this.service = service;
    }

    @GET
    public String emailMeAPoem() {
        return service.writeAPoem("Quarkus", 4);
    }

}
```

then an example trace is:



In the trace above we can see the parent span which corresponds to the handling the GET HTTP request, but the real interesting thing is the `langchain4j.aiservices.MyAiService.writeAPoem` span which corresponds to the invocation of the AI service. The child spans of this span correspond (from to right) to calling the OpenAI API, invoking the `sendEmail` tool and finally invoking calling the OpenAI API again.

Auditing

The extension allows users to audit the process of implementing an AiService by introducing `io.quarkiverse.langchain4j.audit.AuditService` and `io.quarkiverse.langchain4j.audit.Audit`. By default, if a bean of type `AuditService` is present in the application, it will be used in order to create an `Audit`, which received various call-backs pertaining to the implementation of the AiService method. More information can be found on the javadoc of these two classes.
