Spring AI / Reference / Retrieval Augmented Generation (RAG)

Retrieval Augmented Generation

Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) is a technique useful to overcome the limitations of large language models that struggle with long-form content, factual accuracy, and context-awareness.

Spring AI supports RAG by providing a modular architecture that allows you to build custom RAG flows yourself or use out-of-the-box RAG flows using the `Advisor` API.

NOTE

Learn more about Retrieval Augmented Generation in the concepts section.

Advisors

Spring AI provides out-of-the-box support for common RAG flows using the `Advisor` API.

To use the `QuestionAnswerAdvisor` or `RetrievalAugmentationAdvisor`, you need to add the `spring-ai-advisors-vector-store` dependency to your project:

```xml
<dependency>
   <groupId>org.springframework.ai</groupId>
   <artifactId>spring-ai-advisors-vector-store</artifactId>
</dependency>
```

QuestionAnswerAdvisor

A vector database stores data that the AI model is unaware of. When a user question is sent to the AI model, a `QuestionAnswerAdvisor` queries the vector database for documents related to the user question.

The response from the vector database is appended to the user text to provide context for the AI model to generate a response.

Assuming you have already loaded data into a `VectorStore`, you can perform Retrieval Augmented Generation (RAG) by providing an instance of `QuestionAnswerAdvisor` to the `ChatClient`.

```java
ChatResponse response = ChatClient.builder(chatModel)
        .build().prompt()
        .advisors(new QuestionAnswerAdvisor(vectorStore))
        .user(userText)
        .call()
        .chatResponse();
```

In this example, the `QuestionAnswerAdvisor` will perform a similarity search over all documents in the Vector Database. To restrict the types of documents that are searched, the `SearchRequest` takes an SQL like filter expression that is portable across all `VectorStores`.

This filter expression can be configured when creating the `QuestionAnswerAdvisor` and hence will always apply to all `ChatClient` requests, or it can be provided at runtime per request.

Here is how to create an instance of `QuestionAnswerAdvisor` where the threshold is `0.8` and to return the top `6` results.

```java
var qaAdvisor = QuestionAnswerAdvisor.builder(vectorStore)

.searchRequest(SearchRequest.builder().similarityThreshold(0.8d).topK(6).b
uild())
        .build();
```

Dynamic Filter Expressions

Update the `SearchRequest` filter expression at runtime using the `FILTER_EXPRESSION` advisor context parameter:

```java
ChatClient chatClient = ChatClient.builder(chatModel)
    .defaultAdvisors(QuestionAnswerAdvisor.builder(vectorStore)
        .searchRequest(SearchRequest.builder().build())
        .build())
    .build();

// Update filter expression at runtime
String content = this.chatClient.prompt()
    .user("Please answer my question XYZ")
    .advisors(a -> a.param(QuestionAnswerAdvisor.FILTER_EXPRESSION, "type
== 'Spring'"))
    .call()
    .content();
```

The `FILTER_EXPRESSION` parameter allows you to dynamically filter the search results based on the provided expression.

Custom Template

The `QuestionAnswerAdvisor` uses a default template to augment the user question with the retrieved documents. 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 context with the user query. 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:

- a `query` placeholder to receive the user question.
- a `question_answer_context` placeholder to receive the retrieved context.

JAVA

```
PromptTemplate customPromptTemplate = PromptTemplate.builder()

.renderer(StTemplateRenderer.builder().startDelimiterToken('<').endDelimit
erToken('>').build())
    .template("""
            <query>

            Context information is below.

            ---------------------
            <question_answer_context>
            ---------------------

            Given the context information and no prior knowledge, answer
the query.

            Follow these rules:

            1. If the answer is not in the context, just say that you
don't know.
            2. Avoid statements like "Based on the context..." or "The
provided information...".
            """)
    .build();

    String question = "Where does the adventure of Anacletus and Birba
take place?";

    QuestionAnswerAdvisor qaAdvisor =
QuestionAnswerAdvisor.builder(vectorStore)
        .promptTemplate(customPromptTemplate)
        .build();

    String response = ChatClient.builder(chatModel).build()
        .prompt(question)
        .advisors(qaAdvisor)
        .call()
        .content();
```

NOTE

The `QuestionAnswerAdvisor.Builder.userTextAdvise()` method is deprecated in favor of using `.promptTemplate()` for more flexible customization.

RetrievalAugmentationAdvisor

Spring AI includes a library of RAG modules that you can use to build your own RAG flows. The `RetrievalAugmentationAdvisor` is an `Advisor` providing an out-of-the-box implementation for the most common RAG flows, based on a modular architecture.

Sequential RAG Flows

Naive RAG

JAVA

```
Advisor retrievalAugmentationAdvisor =
RetrievalAugmentationAdvisor.builder()
        .documentRetriever(VectorStoreDocumentRetriever.builder()
                .similarityThreshold(0.50)
                .vectorStore(vectorStore)
                .build())
        .build();

String answer = chatClient.prompt()
        .advisors(retrievalAugmentationAdvisor)
        .user(question)
        .call()
        .content();
```

By default, the `RetrievalAugmentationAdvisor` does not allow the retrieved context to be empty. When that happens, it instructs the model not to answer the user query. You can allow empty context as follows.

JAVA

```
Advisor retrievalAugmentationAdvisor =
RetrievalAugmentationAdvisor.builder()
        .documentRetriever(VectorStoreDocumentRetriever.builder()
                .similarityThreshold(0.50)
                .vectorStore(vectorStore)
                .build())
        .queryAugmenter(ContextualQueryAugmenter.builder()
                .allowEmptyContext(true)
                .build())
        .build();

String answer = chatClient.prompt()
        .advisors(retrievalAugmentationAdvisor)
        .user(question)
        .call()
        .content();
```

The `VectorStoreDocumentRetriever` accepts a `FilterExpression` to filter the search results based on metadata. You can provide one when instantiating the `VectorStoreDocumentRetriever` or at runtime per request, using the `FILTER_EXPRESSION` advisor context parameter.

JAVA

```
Advisor retrievalAugmentationAdvisor =
RetrievalAugmentationAdvisor.builder()
        .documentRetriever(VectorStoreDocumentRetriever.builder()
                .similarityThreshold(0.50)
                .vectorStore(vectorStore)
                .build())
        .build();

String answer = chatClient.prompt()
        .advisors(retrievalAugmentationAdvisor)
```

```java
        .advisors(a ->
a.param(VectorStoreDocumentRetriever.FILTER_EXPRESSION, "type ==
'Spring'"))
        .user(question)
        .call()
        .content();
```

See VectorStoreDocumentRetriever for more information.

Advanced RAG

```java
Advisor retrievalAugmentationAdvisor =
RetrievalAugmentationAdvisor.builder()
        .queryTransformers(RewriteQueryTransformer.builder()
                .chatClientBuilder(chatClientBuilder.build().mutate())
                .build())
        .documentRetriever(VectorStoreDocumentRetriever.builder()
                .similarityThreshold(0.50)
                .vectorStore(vectorStore)
                .build())
        .build();

String answer = chatClient.prompt()
        .advisors(retrievalAugmentationAdvisor)
        .user(question)
        .call()
        .content();
```

You can also use the `DocumentPostProcessor` API to post-process the retrieved documents before passing them to the model. For example, you can use such an interface to perform re-ranking of the retrieved documents based on their relevance to the query, remove irrelevant or redundant documents, or compress the content of each document to reduce noise and redundancy.

Modules

Spring AI implements a Modular RAG architecture inspired by the concept of modularity detailed in the paper "Modular RAG: Transforming RAG Systems into LEGO-like Reconfigurable Frameworks".

Pre-Retrieval

Pre-Retrieval modules are responsible for processing the user query to achieve the best possible retrieval results.

Query Transformation

A component for transforming the input query to make it more effective for retrieval tasks, addressing challenges such as poorly formed queries, ambiguous terms, complex vocabulary, or unsupported languages.

IMPORTANT

> When using a `QueryTransformer`, it's recommended to configure the `ChatClient.Builder` with a low temperature (e.g., 0.0) to ensure more deterministic and accurate results, improving retrieval quality. The default temperature for most chat models is typically too high for optimal query transformation, leading to reduced retrieval effectiveness.

CompressionQueryTransformer

A `CompressionQueryTransformer` uses a large language model to compress a conversation history and a follow-up query into a standalone query that captures the essence of the conversation.

This transformer is useful when the conversation history is long and the follow-up query is related to the conversation context.

```java
Query query = Query.builder()
        .text("And what is its second largest city?")
        .history(new UserMessage("What is the capital of Denmark?"),
                new AssistantMessage("Copenhagen is the capital of
Denmark."))
        .build();

QueryTransformer queryTransformer = CompressionQueryTransformer.builder()
        .chatClientBuilder(chatClientBuilder)
        .build();

Query transformedQuery = queryTransformer.transform(query);
```

The prompt used by this component can be customized via the `promptTemplate()` method available in the builder.

RewriteQueryTransformer

A `RewriteQueryTransformer` uses a large language model to rewrite a user query to provide better results when querying a target system, such as a vector store or a web search engine.

This transformer is useful when the user query is verbose, ambiguous, or contains irrelevant information that may affect the quality of the search results.

```java
Query query = new Query("I'm studying machine learning. What is an LLM?");

QueryTransformer queryTransformer = RewriteQueryTransformer.builder()
        .chatClientBuilder(chatClientBuilder)
        .build();

Query transformedQuery = queryTransformer.transform(query);
```

The prompt used by this component can be customized via the `promptTemplate()` method available in the builder.

TranslationQueryTransformer

A `TranslationQueryTransformer` uses a large language model to translate a query to a target language that is supported by the embedding model used to generate the document embeddings. If the query is already in the target language, it is returned unchanged. If the language of the query is unknown, it is also returned unchanged.

This transformer is useful when the embedding model is trained on a specific language and the user query is in a different language.

```java
Query query = new Query("Hvad er Danmarks hovedstad?");

QueryTransformer queryTransformer = TranslationQueryTransformer.builder()
        .chatClientBuilder(chatClientBuilder)
        .targetLanguage("english")
        .build();

Query transformedQuery = queryTransformer.transform(query);
```

The prompt used by this component can be customized via the `promptTemplate()` method available in the builder.

Query Expansion

A component for expanding the input query into a list of queries, addressing challenges such as poorly formed queries by providing alternative query formulations, or by breaking down complex problems into simpler sub-queries.

MultiQueryExpander

A `MultiQueryExpander` uses a large language model to expand a query into multiple semantically diverse variations to capture different perspectives, useful for retrieving additional contextual information and increasing the chances of finding relevant results.

```java
MultiQueryExpander queryExpander = MultiQueryExpander.builder()
    .chatClientBuilder(chatClientBuilder)
    .numberOfQueries(3)
    .build();
List<Query> queries = queryExpander.expand(new Query("How to run a Spring
Boot app?"));
```

By default, the `MultiQueryExpander` includes the original query in the list of expanded queries. You can disable this behavior via the `includeOriginal` method in the builder.

```java
MultiQueryExpander queryExpander = MultiQueryExpander.builder()
    .chatClientBuilder(chatClientBuilder)
    .includeOriginal(false)
    .build();
```

The prompt used by this component can be customized via the `promptTemplate()` method available in the builder.

Retrieval

Retrieval modules are responsible for querying data systems like vector store and retrieving the most relevant documents.

Document Search

Component responsible for retrieving `Documents` from an underlying data source, such as a search engine, a vector store, a database, or a knowledge graph.

VectorStoreDocumentRetriever

A `VectorStoreDocumentRetriever` retrieves documents from a vector store that are semantically similar to the input query. It supports filtering based on metadata, similarity threshold, and top-k results.

```java
DocumentRetriever retriever = VectorStoreDocumentRetriever.builder()
    .vectorStore(vectorStore)
    .similarityThreshold(0.73)
    .topK(5)
    .filterExpression(new FilterExpressionBuilder()
        .eq("genre", "fairytale")
        .build())
    .build();
```

```java
List<Document> documents = retriever.retrieve(new Query("What is the main
character of the story?"));
```

The filter expression can be static or dynamic. For dynamic filter expressions, you can pass a `Supplier`.

```java
DocumentRetriever retriever = VectorStoreDocumentRetriever.builder()
    .vectorStore(vectorStore)
    .filterExpression(() -> new FilterExpressionBuilder()
        .eq("tenant", TenantContextHolder.getTenantIdentifier())
        .build())
    .build();
List<Document> documents = retriever.retrieve(new Query("What are the KPIs
for the next semester?"));
```

You can also provide a request-specific filter expression via the `Query` API, using the `FILTER_EXPRESSION` parameter. If both the request-specific and the retriever-specific filter expressions are provided, the request-specific filter expression takes precedence.

```java
Query query = Query.builder()
    .text("Who is Anacletus?")
    .context(Map.of(VectorStoreDocumentRetriever.FILTER_EXPRESSION,
"location == 'Whispering Woods'"))
    .build();
List<Document> retrievedDocuments = documentRetriever.retrieve(query);
```

Document Join

A component for combining documents retrieved based on multiple queries and from multiple data sources into a single collection of documents. As part of the joining process, it can also handle duplicate documents and reciprocal ranking strategies.

ConcatenationDocumentJoiner

A `ConcatenationDocumentJoiner` combines documents retrieved based on multiple queries and from multiple data sources by concatenating them into a single collection of documents. In case of duplicate documents, the first occurrence is kept. The score of each document is kept as is.

```java
Map<Query, List<List<Document>>> documentsForQuery = ...
DocumentJoiner documentJoiner = new ConcatenationDocumentJoiner();
List<Document> documents = documentJoiner.join(documentsForQuery);
```

Post-Retrieval

Post-Retrieval modules are responsible for processing the retrieved documents to achieve the best possible generation results.

Document Post-Processing

A component for post-processing retrieved documents based on a query, addressing challenges such as *lost-in-the-middle*, context length restrictions from the model, and the need to reduce noise and redundancy in the retrieved information.

For example, it could rank documents based on their relevance to the query, remove irrelevant or redundant documents, or compress the content of each document to reduce noise and redundancy.

Generation

Generation modules are responsible for generating the final response based on the user query and retrieved documents.

Query Augmentation

A component for augmenting an input query with additional data, useful to provide a large language model with the necessary context to answer the user query.

ContextualQueryAugmenter

The `ContextualQueryAugmenter` augments the user query with contextual data from the content of the provided documents.

```java
QueryAugmenter queryAugmenter =
ContextualQueryAugmenter.builder().build();
```

By default, the `ContextualQueryAugmenter` does not allow the retrieved context to be empty. When that happens, it instructs the model not to answer the user query.

You can enable the `allowEmptyContext` option to allow the model to generate a response even when the retrieved context is empty.

```java
QueryAugmenter queryAugmenter = ContextualQueryAugmenter.builder()
        .allowEmptyContext(true)
        .build();
```

The prompts used by this component can be customized via the `promptTemplate()` and `emptyContextPromptTemplate()` methods available in the builder.