Building a RAG application with Redis and Spring AI

Introduction

Vector databases are crucial for AI applications, especially when it comes to performing similarity searches instead of exact matches. This distinction is key for applications where finding similar items or documents is more relevant than precise matches, which is a common scenario in recommendation systems and natural language processing tasks. Redis, a popular in-memory data store, has gained traction as a reliable option for vector databases due to its performance and ease of use.

The Spring community recently introduced a new project called Spring AI, which aims to simplify the development of AI-powered applications, including those that leverage vector databases. In this article, we will demonstrate how to build a Spring AI application that utilizes Redis as the vector database, focusing on implementing a Retrieval Augmented Generation (RAG) workflow.

Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) is a technique used to integrate data with AI models. In a RAG workflow, the first step involves loading data into a vector database, such as Redis. When a user query is received, the vector database retrieves a set of documents similar to the query. These documents then serve as the context for the user's question and are used in conjunction with the user's query to generate a response, typically through an AI model.

In our demonstration, we will use a dataset containing information about beers, including attributes such as name, Alcohol By Volume (ABV), International Bitterness Units (IBU), and a description for each beer. This dataset will be loaded into Redis to demonstrate the RAG workflow.

Data Load

The data we will use for our application consists of JSON documents providing information about beers. Each document has the following structure:

```json
{
  "id": "00gkb9",
  "name": "Smoked Porter Ale",
  "description": "The Porter Pounder Smoked Porter is a dark rich flavored ale that is made with 5 malts that include smoked and chocolate roasted malts. It has coffee and mocha notes that create a long finish that ends clean with the use of just a bit of dry hopping",
  "abv": 8,
  "ibu": 36
}
```

To load this beer dataset into Redis, we will use the RagDataLoader class. This class contains a run method that is executed at application startup. Within this method, we use a JsonReader to parse the dataset and then insert the documents into Redis using the autowired VectorStore.

```
JsonReader jsonLoader = new JsonReader(resource, "name", "abv", "ibu", "description");
①
vectorStore.add(jsonLoader.get()); ②
```

① Create a JsonReader with fields relevant to our use case

② Use the autowired VectorStore to insert the documents into Redis

What we have at this point is a dataset of about 20,000 beers with their corresponding embeddings.

RAG Service

The RagService class implements the RAG workflow. When a user prompt is received, the retrieve method is called, which performs the following steps:

- Computes the vector of the user prompt
- Queries the Redis database to retrieve the most relevant documents
- Constructs a prompt using the retrieved documents and the user prompt
- Calls a ChatClient with the prompt to generate a response

```
public Generation retrieve(String message) {
   SearchRequest request = SearchRequest.query(message).withTopK(topK);
   List<Document> similarDocuments = vectorStore.similaritySearch(request); ①
   Message systemMessage = getSystemMessage(similarDocuments);
   UserMessage userMessage = new UserMessage(message);
   Prompt prompt = new Prompt(List.of(systemMessage, userMessage)); ②
   ChatResponse response = chatClient.call(prompt); ③
   return response.getResult();
}
```

① Query Redis for the top K documents most relevant to the input message

② Assemble the complete prompt using a template

③ Call the autowired ChatClient with the prompt

Controller

Now that we have implemented our RAG service we can wrap it in a HTTP endpoint.

The RagController class exposes it as a POST endpoint:

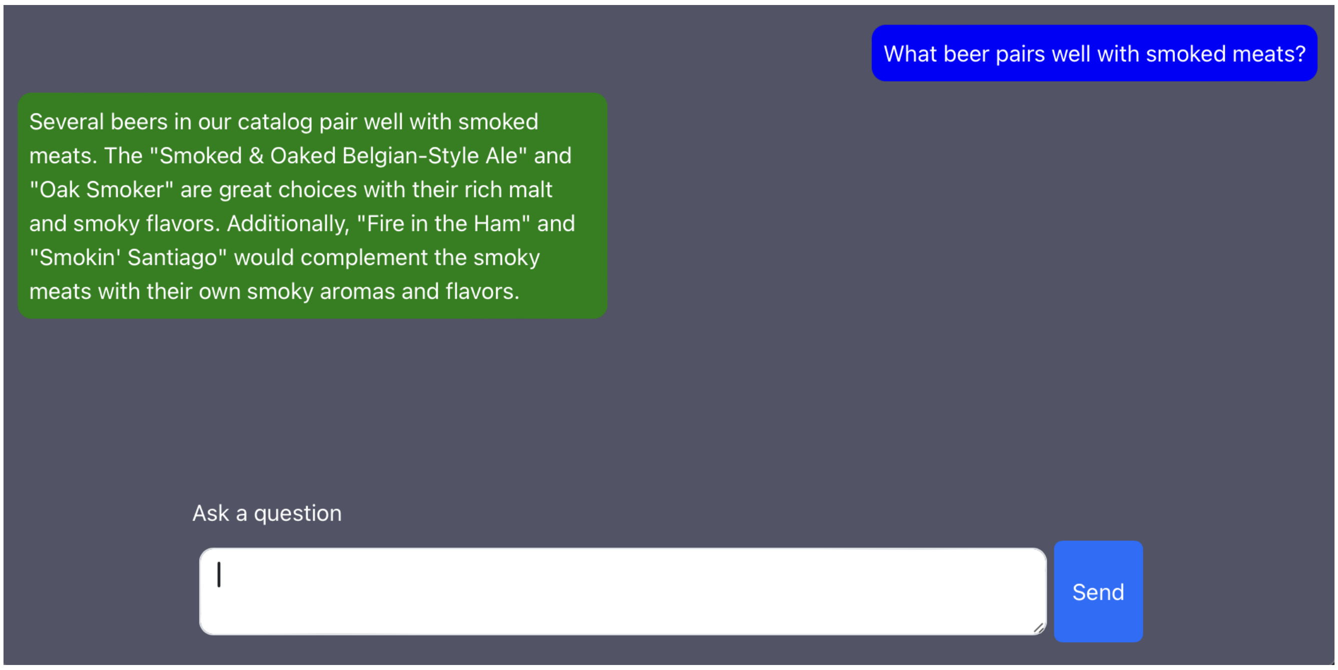
```
@PostMapping("/chat/{chatId}")
@ResponseBody
public Message chatMessage(@PathVariable("chatId") String chatId, @RequestBody Prompt
prompt) {
    Generation generation = ragService.retrieve(prompt.getPrompt()); ①
    return Message.of(generation.getOutput().getContent()); ②
}
```

① Extract the user prompt from the body and pass it to the autowired `RagService`

② Reply with the generated message

User Interface

For the user interface, we have created a simple React frontend that allows users to ask questions about beers. The frontend interacts with the Spring backend by sending HTTP requests to the `/chat/{chatId}` endpoint and displaying the responses.



What beer pairs well with smoked meats?

Several beers in our catalog pair well with smoked meats. The "Smoked & Oaked Belgian-Style Ale" and "Oak Smoker" are great choices with their rich malt and smoky flavors. Additionally, "Fire in the Ham" and "Smokin' Santiago" would complement the smoky meats with their own smoky aromas and flavors.

Voilà! With just a few classes we have implemented a RAG application with Spring AI and Redis.

Related Resources

- The code presented in this article is available on GitHub.
- For more information about Spring AI, visit the project homepage.
- Learn more about the Redis vector search API in the Redis vector documentation.