Buid your AI assistant with Ollama, LLM and Spring

Table of Contents

Chapter 1. Introduction

1.1. And so we meet. Let’s get to know each other…

Hey there! You have probably read the title and the description of this workshop, but to make sure we are on the same page, I’ll try to answer some questions in this chapter.

What is this workshop for?

Obviously, building an AI assistant…

Okay, this is not a good explanation. Let’s break it down a bit.

The idea of this workshop is to show you how you can utilize the power of Large Language Models (LLMs) in your web application and make it even more flexible. There are a plethora of use cases where you can combine LLMs with the business logic of a server application, with only your imagination as the limitation.

Unlike simple tutorials that show API calls, this workshop will take a practical, hands-on approach. By the end, you will have a working web application that leverages an LLM for real-world tasks.

How does this suit me?

This suits you in many ways. Have you ever thought of a scenario where your application has a lot of input data that needs a human-like approach to analyze? Let’s look at some examples:

Example 1:
You’re building an app that tracks people’s fitness and eating habits. The app monitors metrics like the person’s bodily parameters (weight, height, age, gender, etc.), and it also tracks their eating habits, including the foods and meals they consume, caloric intake, nutritional levels, and so on. Usually, you would need someone to enter the data of each food, determine its nutrient content, and add logic to combine these parameters in a scientifically proven method that will inform the user about their health choices.

The problem?
You can’t manage all the foods in the world and find out all the information that your app needs to provide proper information to the user. Also, one size does not fit all. What if you needed to output information that is more targeted to the user?

The solution?
You integrate an LLM into your application. You add a specific prompt, customized to the user’s needs, and ask it to return the appropriate data. The LLM can be used to look up nutritional data and process health insights tailored to the user’s needs. The data can also be saved to the database to be reused for other users who might have the same requirements.

Example 2:
You build librarian software. Your application keeps track of all the books contained within a library. It also knows where each book can be found. Your application searches for the right book, tracks book rentals, and helps users navigate to the right section.

The problem?
What if you wanted your application to do a bit more? For example, give appropriate recommendations to the user based on a description of what they are looking for, or let the app recommend books from your library to that user based on their ratings of previous books?

The solution?
Use an LLM. You can create specific LLM prompts that include the user's description of the book, their preferences, and ratings of previous readings. The LLM could be trained on book knowledge, and you can even implement logic where the LLM interacts with your application, asking if this book is in the library or not, so it can build up a list of the books that are only present within your library.

Example 3:

You are building a home assistant server capable of managing your smart devices. The server's API accepts specific intents that trigger hardcoded scenarios to switch on/off lights, collect data from sensors, etc.

The problem?
You want your server to be smarter. You want your intents to be more flexible. For example, when you say "Lights on!" to your voice assistant, you want it to turn on certain lights, depending on the time and atmosphere you desire.

The solution?
You can integrate an LLM into the server. Your server will forward the voice input from a third-party assistant like Google Home or Alexa and prompt your LLM to identify the intent and query the right devices. The LLM will then return a proper response that is read by the home assistant and will trigger the smart devices that it determines need to be activated according to the prompt.

To keep this section brief, I am providing vague examples here. All of these will have their specific constraints and challenges, but none of them are impossible. You can think of any task that requires more abstract thinking and assign an LLM to do it for you.

Okay! I'm convinced! Where do we go from here?

To start off, you're in the right place! In this workshop, we are going to build such a solution by delving deeper into one of the aforementioned examples. We will build a web application server that will gather input from the user and consult an LLM for the things that we can't or

don’t want to figure out on our own. For that purpose, I am going to show you how to implement the first example.

We will build an application that gathers information about the eating habits of a user and uses an LLM to determine nutritional facts about the food they consume. As a result, we will provide each user with proper information about their eating habits and generate a review based on the items they consumed. This will help the user make better and healthier food choices that are more in tune with their personal bodily parameters.

What’s the technology stack we’re going to use?

The technology stack is straightforward and requires minimal setup on your side. Here is what we’re going to use:

- **Spring Boot framework** - the core of our application, handling API requests and processing data.
- **PostgreSQL database** - stores user input and processed nutritional data from the LLM.
- **Ollama server** - a locally hosted web server that enables API communication with an LLM model.
- **Llama LLM** - the chosen LLM for this workshop, due to its lightweight models that can run on most modern machines.
- **Docker** - used to simplify setup by providing a Docker Compose script for database and Ollama server management.

We are not doing fine-tuning in this workshop. Instead, we will focus on prompt engineering and API-based interaction with pre-trained LLMs.

But isn’t running an LLM locally slow? (The system requirements)

Running an LLM locally can be resource-intensive, but for the small-scale tasks in this workshop, most modern machines should handle it fine.

Recommended system requirements:

- **A computer running macOS, Linux, or Windows (with WSL installed!)**
- **At least 16GB of RAM** - LLMs are memory-intensive.
- **A modern CPU** - the more cores, the better.
- **At least 50GB of free disk space** - required for the LLM model, Docker images, and dependencies.
- **A dedicated GPU** - optional, but it significantly improves performance if properly configured.

Now that we’re all set, let’s move to the next chapter, where I will take you through our project

design and show you the big picture. :imagesdir: img :source-highlighter: coderay :icons: font

Chapter 2. Project Structure

Before we begin doing anything, we first need to get a better understanding of what we're building. This section is dedicated just for that. We will start with a diagram of what our project will look like.

2.1. Component diagram

Simply put our component diagram will look like this:

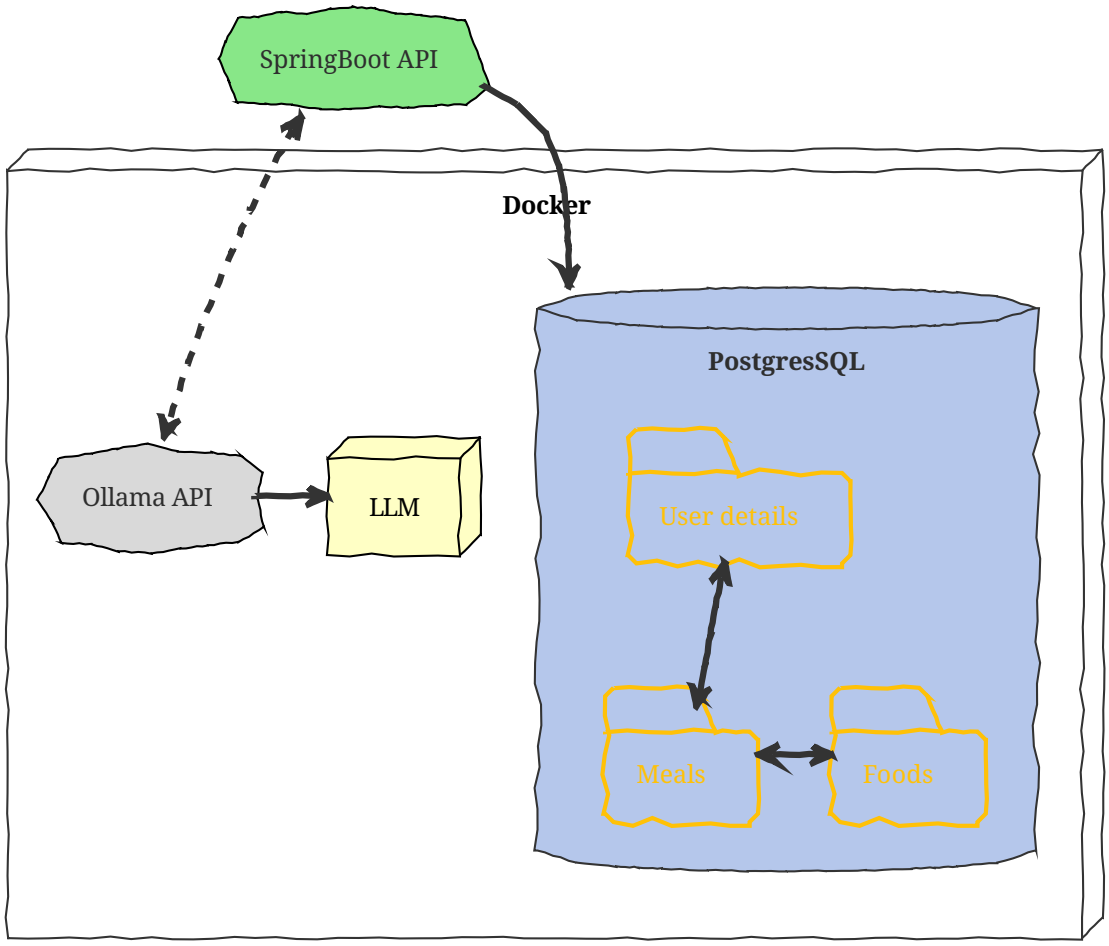


Figure 1. Component diagram

What we see here is a complicated illustration of components called technological stack.

- **SpringBoot API server** - this is the heart of webapp service. It will contain the endpoints that will interact with our users and will compute the prompts and process the responses from our LLM.
- **Ollama API server** - this is our gateway to the LLM. With the powerful plugins of the Spring framework, we will be able to easily access those APIs, by simply adding a config line and setting up some methods inside our code.
- **LLM model** - the brain of our advanced web application. It will take the prompt and the input of Ollama, do it's magic and spit out results in the format that the prompt has specified.
- **PostgresSQL** - we need to store all tha generated data by the user and the LLM somewhere, so

we can easily recall it and spare calls to the LLM. If we receive a response once, we can save it and reuse it, instead of asking the LLM to compute it again. This is costing time and computational power after all.

- **Docker** - to make our lives easier, we will wrap the database, Ollama server and the LLM into docker containers. This will help us set up those components with a simple script, rather than having to read long tutorials on how to install each stuff separately. It will also help us pay more attention to the code, instead of trying to figure out "Why it doesn't work on my machine?"

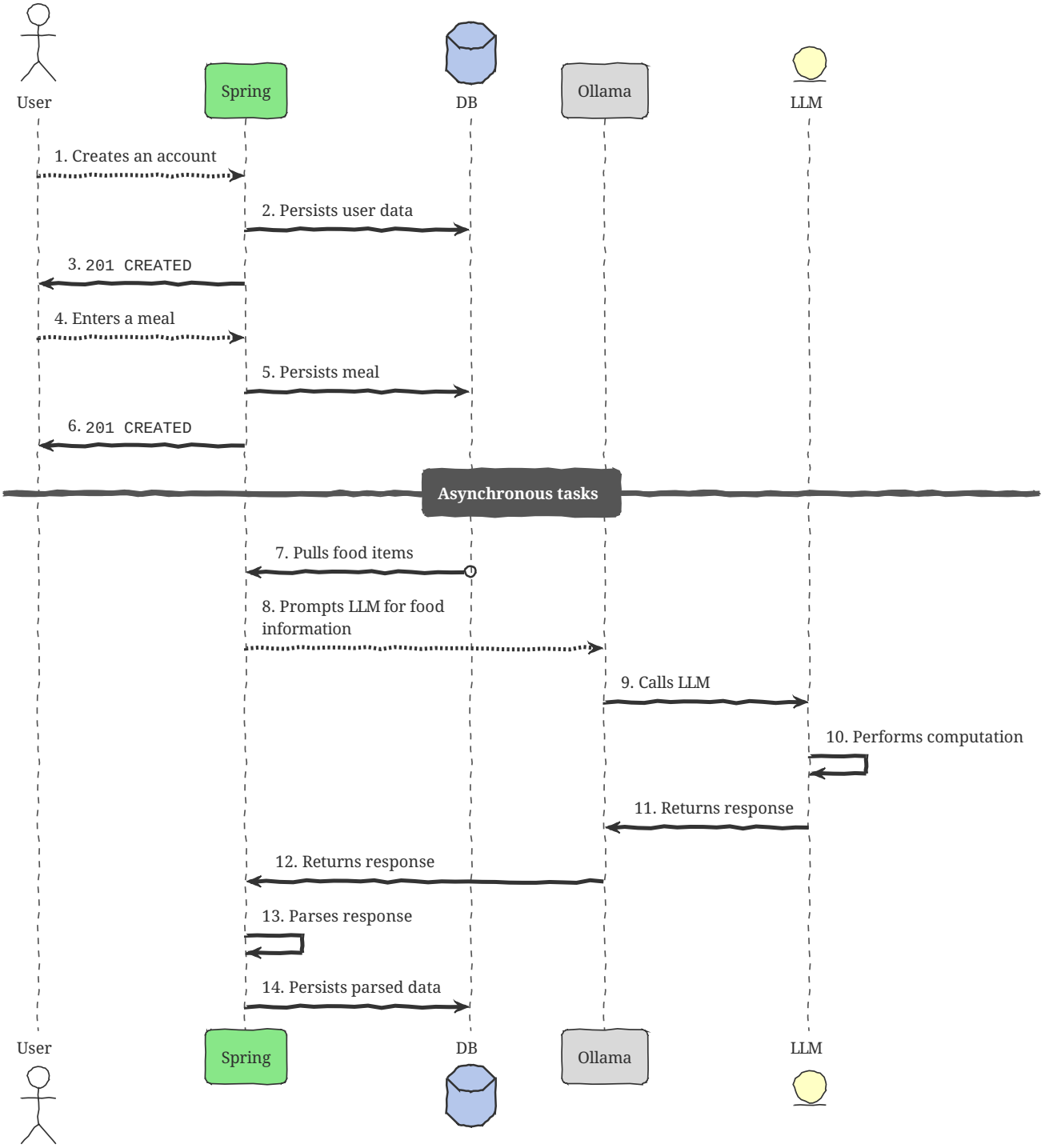
Keep in mind that this is a very general component diagram, intended to give you general understanding on how our project will look like. All the underlying specifics will be examined throughout each chapter.

2.2. Use cases

Knowing all the components leads us to taking a look at the use cases of this project. This will help us understand what exactly are we doing and how it all comes together.

2.2.1. Registering information

When the user interacts with the app for the first time, the database will be empty, so we might need to query the LLM more often. Once some data has been generated, when the user repeats their requests, our database will already have some of the items we asked the LLM previously for, and will skip those prompts.



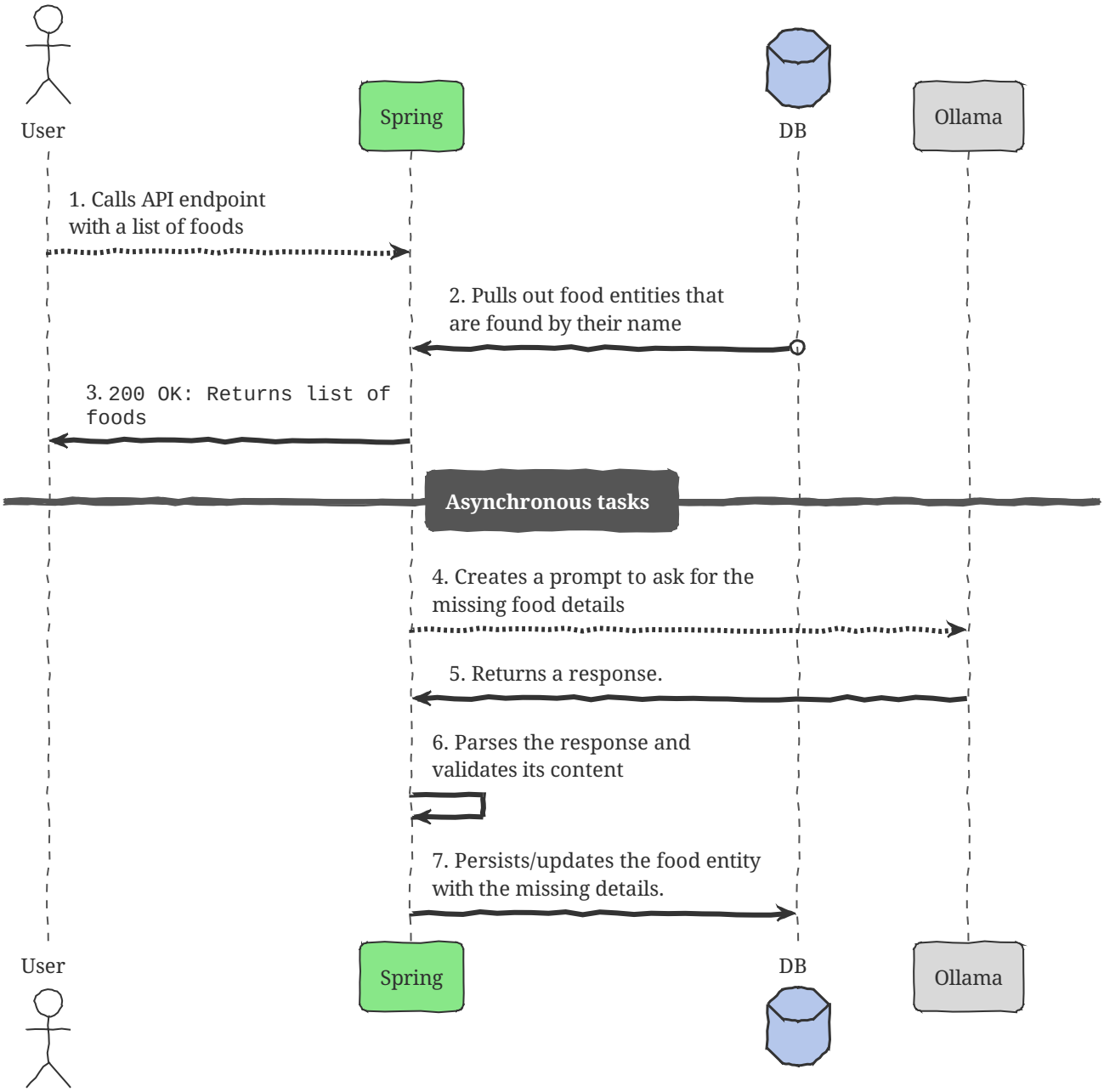
1. The user registers their account. It may contain things like their name age and gender, along with their height, weight and recommended daily caloric intake.
2. The data is consumed by the API, and it is persisted into our database.
3. We send a response to the user that the data has been saved, so they can input their meal information.
4. The user calls an API endpoint that registers a new meal on the database. (e.g.: [5x□, 1x□, 3x□])
5. The data is being processed and persisted.
6. The server sends a response to the user that the data has been saved.
7. The server pulls out the persisted data for the user and checks if any food has missing details. This could be things as calories by quantity and amount, nutritional value, allergens etc.

8. The server builds a prompt for the LLM, telling it what task it has to perform. Along with that it sends a list of foods, that are new to the database.
9. Ollama receives the prompt and the input and formats the query to the LLM.
10. The LLM takes the input and does its magic.
11. The LLM returns a response, that is complying to the prompt, hopefully.
12. The response is formatted to the Ollama's API and returned back to our Spring server.
13. The server takes the response, that is usually a string and parses the string. Depending on the desired response we might want to break down the response and persist each value into a different entity. Sometimes the LLM might not be able to respond properly or return information that we do not want to save. We need to have logic to handle those scenarios.
14. After the server breaks down the response and validates all the conditions, it persists the received data into its own entity.

Ideally we want to delegate LLM tasks to an asynchronous thread, as LLM computation is slow, and we don't want the user to wait for a response. That's why all the tasks that involve LLM will be executed asynchronously.

2.2.2. Requesting food information

Sometimes the user might want to know what are the nutritional values of certain foods. This usecase will show how such a scenario can be handled with the help of an LLM.



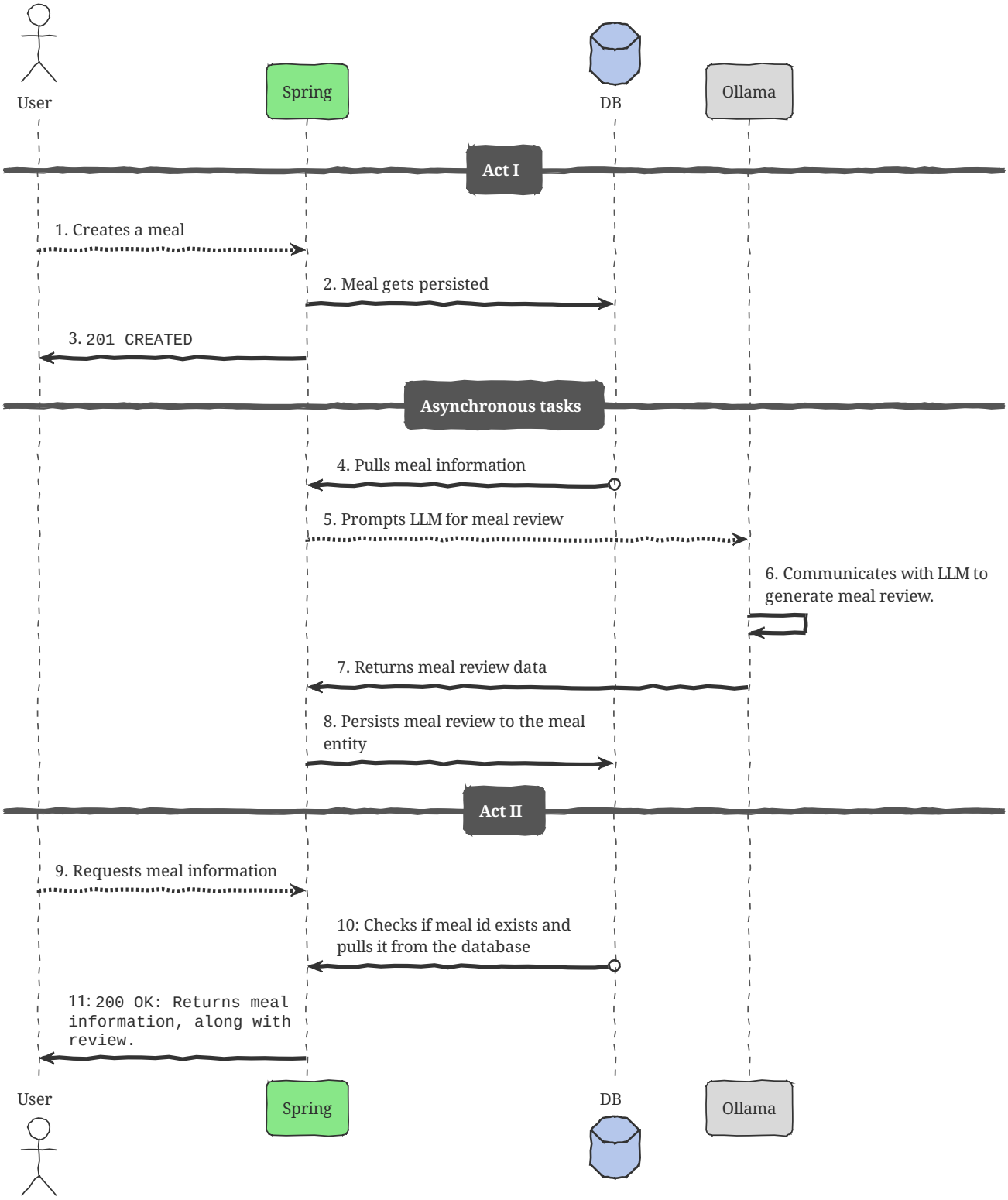
1. The user sends a query with list of food items they want to get information about. (e.g.: [□, □, □]).
2. The server takes those foods and checks inside the database, whether there is data already available.
3. Once the extracted data is gathered, it gets formatted and sent to the user as a response. If any food data is missing, the user will be informed that the data is being generated, and they need to query the food list again later.
4. The server takes the missing food information and creates a prompt to the LLM. That prompt could be just one, containing the list of all the missing foods with a request to return info about the nutritional values, or it can be a single request per food. Deciding how to build a prompt at this step requires a balanced approach. If we try to get all the food information with one prompt, we might get a response that is harder to parse, but on the other hand, if we decide to use separate prompts, we are going to need more time to generate the info for each food item.
5. The data gets processed by the LLM, and Ollama returns the response, that was generated by it.
6. Again, since the response is in string format, we need to parse it and validate it, so we make

sure we don’t persist malformed data to our entities.

7. The gets persisted. Next time the user queries those foods, they will get a response containing their nutritional value.

2.2.3. Asking for meal review

Once the user has created a meal, they might want to know how the food they ate, impacts their health and nutrition. The large language model is just the thing for that! It can take the information, along with the user’s body parameters and generate a review, along with advices how the meal can be improved to be more balanced and nutritious.



1. The user enters their meal info. [6x□, 1x□, 0.5x□]
2. The data gets persisted to the Meal and Foods entities.
3. The user receives a response promptly.
4. The server pulls the newly created meal data from the database (or from a memory source).
5. A prompt is built, specific to the user's attributes (their age, gender, weight, height) along with the food they ate.
6. The LLM is tasked to create a review, with information such as how good the meal is, and

breakdown on what is good/bad and how it can be improved.

7. The generated review is passed back to the application server.
8. The data is persisted to the meal entity.
9. The user requests meal information, by the meal id.
10. The server queries the database for a meal with that id.
11. The newly generated response will have information about the meal details and also will contain the review. If the review is not yet generated, it could state that it is being generated.

These are couple of simple use cases, which clearly demonstrate how an LLM can be applicable in such a scenario. But that's not the limit. You can take what you learned from here and extend the project, inventing other usecase the author of this article couldn't think of. Remember - imagination is the limit!