Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Programarea Aplicatiilor Distribuite

Laboratory work #1

Web Proxy

Author:
Name Bitca Dina
std. gr. FAF-201

Supervisor:
Volosenco Maxim

Chișinău 2023

1 Tasks

1. Assess Application Suitability;
2. Define Service Boundaries;
3. Choose Technology Stack and Communication Patterns;
4. Design Data Management;
5. Set Up Deployment and Scaling;

2 Results

The topic I have chosen in the scope of this laboratory work is the development of a system that would automatically suggest recipes based on what ingredients the user possesses.

2.1 Application Suitability

Developing a system that automatically suggests recipes based on the ingredients a user possesses is a relevant and useful idea with several compelling reasons for its implementation through distributed systems. There are a few reasons that support my idea on this manner:

1. Resource optimization:
   Utilizing a distributed system allows for efficient utilization of computing resources, ensuring the system can handle a large number of users and complex queries simultaneously. An example from real life would be **Netflix**, a very big application that employs a microservices architecture to efficiently handle millions of users and deliver personalized content recommendations based on viewing history and preferences.

2. Scalability and Flexibility:
   A distributed system facilitates easy scaling to accommodate an increasing number of users and adapt to varying loads, ensuring the system remains responsive and available. For example, **Airbnb** employs microservices to handle a large and growing user base, providing a scalable platform for accommodation booking with real-time recommendations.

3. Customization and Personalization:
   Distributed systems enable the customization of recipe suggestions based on individual user preferences, dietary restrictions, and past choices. A good real life example would be constituted by **Spotify** which utilizes microservices to offer personalized music recommendations based on listening history and preferences.

4. Efficient Data Processing:
   A distributed system can handle extensive data processing tasks efficiently, such as analyzing user-provided ingredient lists and matching them with a vast recipe database. Another example would be **Uber**, which utilizes microservices to process real-time data from multiple sources to match riders with nearby drivers and optimize ride-sharing routes.

5. Resilience and Fault Tolerance:
   A distributed system enhances system resilience by isolating faults and failures to specific services, preventing a single point of failure from affecting the entire system. **Amazon** employs a microservices architecture for its e-commerce platform, ensuring resilience and availability, even during high-traffic events like Prime Day.

6. Interoperability and Integration:
   Distributed systems allow for easy integration with other applications and platforms, enhancing the overall value and usability of the recipe suggestion system. Last but not least, another good example is constituted by **PayPal**, which utilizes microservices to integrate various financial services and ensure smooth transactions across different platforms and devices.

In summary, developing an automated recipe suggestion system using distributed systems offers benefits such as enhanced user experience, resource optimization, scalability, customization, efficient data processing, resilience, and interoperability, all of which are demonstrated through successful implementations in real-world projects utilizing microservices.

2.2 Service Boundaries

This system is designed to be simple and efficient, focusing on the core functionality of suggesting recipes based on user-provided ingredients. Each microservice has a clear and specific role, contributing to the overall functionality of the system.

1. **User Interface / API Gateway** allows users to input their ingredients and receive recipe suggestions, while also routing requests from the user interface to the appropriate microservices.

2. **Ingredient Service** manages information about ingredients and allows users to add new ingredients.

3. **Recipe Service** manages the recipe database and supports searching for recipes based on ingredients.

Please see the initial architecture in the image provided below.

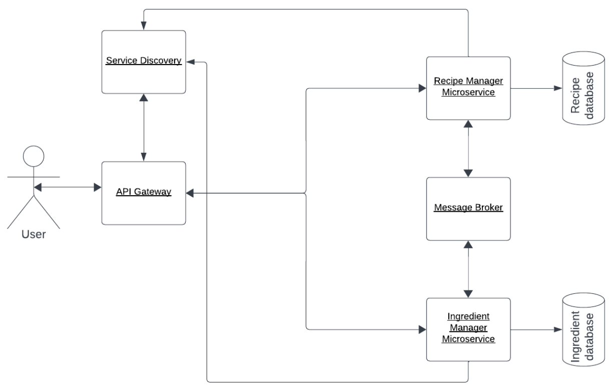

Figure 1: System Architecture

2.3 Technology Stack and Communication Patterns

In the scope of this project, I have chosen to work with the languages Java and C. Java would be used for microservice development, taking into consideration the following reasons.

1. Java has widely adopted microservice frameworks, such as Spring Boot, which is not entirely new to me as a developer;

2. Java has a good support for multithreading and concurrency, which will help me develop scalable microservices that will handle a large number of concurrent requests;

Similarly, I will use C to develop the API Gateway / User interface for the following reasons:

1. C is integrated with .NET Core and ASP.NET, which are powerful and efficient frameworks for building APIs;

2. ASP.NET Core can be used to build API Gateway components with features like request routing, request/response modification, rate limiting, authentication, and load balancing;

3. Learning a new programming language will be a very beneficial and educational experience for me as an aspiring developer;

Given that my project is involving a recipe generator based on ingredients provided by the user, a synchronous communication approach using RESTful APIs is probably the best choice, since the user inputs their available ingredients and expects immediate recipe suggestions based on the input. However, I do not exclude the idea that, while the recipe generator should use synchronous communication for the immediate recipe suggestions, the overall system will most probably use a combination between synchronous and asynchronous communication for other functionalities, such as ingredient or recipe management. I do not exclude the possibility that the communication approach will be updated through the development process.

2.4 Design Data Management

Ingredient Service Endpoints:

1. Add ingredient
   - Type: POST
   - URL: ”/ingredient”
   - Request JSON:

```
{
  ”name”: ”ingredient name”,
}
```

   - Response JSON:

```
{
  ”id”: ”unique−ingredient−id”,
  ”name”: ”ingredient name”
}
```

2. Display all ingredients
   - Type: GET
   - URL: ”/getingredient”
   - Response JSON:

```
{
  ”ingredients”: [
    {
      ”id”: ”unique−ingredient−id”,
      ”name”: ”ingredient name”
    },
    ...
  ]
}
```

3. Get ingredient by ID
   - Type: GET
   - URL:”/getingredient/id”
   - Response JSON:

```
{
  ”id”: ”unique−ingredient−id”,
  ”name”: ”ingredient name”
}
```

Recipe Service Endpoints

1. Add Recipe

   - Type: POST
   - URL: "/recipes"
   - Request JSON:

```
{
  "name": "recipe name",
  "ingredients": ["ingredient-id-1", "ingredient-id-2"],
  "instructions": "Recipe instructions..."
}
```

   - Response JSON:

```
{
  "id": "unique-recipe-id",
  "name": "recipe name",
  "ingredients": ["ingredient-id-1", "ingredient-id-2"],
  "instructions": "Recipe instructions..."
}
```

2. Get All Recipes

   - Type: GET
   - URL: "/recipes"
   - Response JSON:

```
{
  "recipes": [
    {
      "id": "unique-recipe-id",
      "name": "recipe name",
      "ingredients": ["ingredient-id-1", "ingredient-id-2"],
      "instructions": "Recipe instructions..."
    },
    ...
  ]
}
```

3. Get Recipe by ID:

   - Type: GET
   - URL: "/recipes/id"
   - Response JSON:

```
{
  "id": "unique-recipe-id",
  "name": "recipe name",
  "ingredients": ["ingredient-id-1", "ingredient-id-2"],
  "instructions": "Recipe instructions..."
}
```

4. Search Recipes by Ingredients:

   - Type: POST

- URL: "/recipes/search"
- Request JSON:

```
{
  "ingredientIds": ["ingredient-id-1", "ingredient-id-2"]
}
```

- Response JSON:

```
{
  "recipes": [
    {
      "id": "unique-recipe-id",
      "name": "recipe name",
      "ingredients": ["ingredient-id-1", "ingredient-id-2"],
      "instructions": "Recipe instructions..."
    },
    ...
  ]
}
```

2.5 Deployment and Scaling

Considering the nature of my project, which involves a microservices-based recipe suggestion system, both containerization using Docker and orchestration using Kubernetes can be highly beneficial for deployment and scaling. The advantages of using Docker are:

- Isolation and Portability
  Docker provides a lightweight, isolated environment for each microservice, ensuring consistency and portability across different environments.
- Ease of Deployment
  Docker containers can be quickly deployed, making the deployment process efficient and consistent.
- Resource Optimization
  Docker containers share the host OS kernel, leading to efficient resource utilization compared to traditional virtualization.

On the other hand, the advantages of using Kubernetes are:

- Automated Orchestration
  Kubernetes automates the deployment, scaling, and management of containerized applications, allowing for seamless scaling and self-healing capabilities.
- Service Discovery and Load Balancing
  Kubernetes handles service discovery and load balancing, crucial for managing the multiple instances of microservices.
- Scaling
  Kubernetes offers horizontal scaling, allowing you to scale individual microservices based on demand.

Overall, my chosen deployment strategy is containerization using Docker, I chose to containerize each microservice using Docker to encapsulate them with their dependencies and runtime environment.

References

[1] Microservices Architecture, *https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9*

[2] Microservices and Microservices Tools, *https://middleware.io/blog/microservices-architecture/*