Systems Integration Assignment 2 Reactor Core/Web Reactive

Adelino Pais*, Diogo Alves*
**University of Coimbra, DEI*
Emails: acp@student.dei.uc.pt
ddalves@student.dei.uc.pt
Date:10/11/2023

I. Introduction

This assignment focuses on creating a web application using WebFlux and reactive programming based on the Spring Boot framework. It entails building a server that exposes web services and a client application to consume these services, specifically for managing pet and owner data.

The server offers basic CRUD operations for pets and owners. To maintain simplicity, no enhancements were made to the server. Instead, it provides pet identifiers based on owner IDs.

On the client side, the objective is to collect and process data from the server, following specific queries. Responses are stored in separate files and can run in parallel. We aimed to minimize blocking points and handle network failures with up to three connection retries.

This report details our development journey, challenges, solutions, and the outcomes of creating a reactive web application with WebFlux. It covers the server and client components, key functionalities, and techniques used to achieve our objectives.

II. Project Structure

The server developed on the assignment represents the back-end web application built on Spring Boot and WebFlux. It revolves around managing pet and owner data stored in a PostgreSQL database. Here's an overview of the server's structure:

1. Model Classes (Pet and Owner): These classes define data model for pets and owners.

2. Repositories (OwnerRepository and PetRepository): These interfaces serve as the bridge between the application and the database.

3. Services (OwnerService and PetService): Responsible for the business logic, these classes orchestrate tasks related to owners and pets.

4. Controllers (OwnerController and PetController): These components expose REST API endpoints, facilitating interactions with the server.

5. Application Configuration (ServerSpringApplication): This class initiates the Spring Boot application, configures the WebFlux framework, and establishes the database connection.

6. Application Properties (application.properties): This configuration file centralizes essential properties, including the database connection details.

The client application is designed to consume web services provided by the server and perform various queries on pet and owner data. It interacts with the server through a WebClient to retrieve and process the information. Here's an overview of the client's structure:

1. Services (OwnerService and PetService): The client application has two main service classes, "OwnerService" and "PetService." These services are responsible for making requests to the server, handling responses, and processing data.

2. Main Application and WebClient Integration (ClientApplication): This class acts as the main application and integrates the WebClient for making requests to the server.

3. Configuration (application.properties): This file includes basic configuration for the client application.

III. Project Details

In this section, we delve into the workings of both the server and client components of the project.

A. Server Details

The main objective of the server is to receive and send a response to a request based on its endpoint. In the project, for each model class (pet and owner), 5 different CRUD endpoints were created. The first one is for creating either a pet or an owner, the second one is for returning the data of all pets or owners, the third one is for returning the data of a specific pet or owner, the fourth one is for updating the data of a specific pet or owner, and the last one is for deleting a specific owner or pet.

Each endpoint method, when triggered, invokes a method from the respective server, where they interact with the repository to perform the specific action on the database.

Since the repositories created already extend from a repository that contains the basic CRUD operations, most of the methods used in the services to manipulate the database were not defined in their respective repositories. The exception is the method "findByOwnerid(int id)," which is used to determine if an owner is associated with a pet in case we want to eliminate him.

Log entries were also created in the service classes to send information to the console, allowing us to see if the requests were performed successfully or failed. The log's configuration is defined in the ”log4j2.xml” file.

Additionally, an SQL schema file was created to be executed every time the server runs. This file creates the ”pet” and ”owner” tables with the necessary constraints and inserts some data into them.

Finally, all the database connection details are defined in the ”application.properties” file.

B. Client Details

The primary goal of the client application is to consume the web services provided by the server and manipulate the received data to fulfill the requested queries.

To better organize these queries, two services were created for each entity.

In the first query, the client retrieves all the owners from the database. Since we only need two specific attributes, we use the ”map” operator to transform the data into a string containing only the name and number of the owner.

For the second query, the client retrieves all the pets and uses the ”map” operator to determine the size of the JSON array that contains the pets.

In the third query, the client retrieves all the pets and filters them to return only those with the species ”dog.” Afterward, the ”count” operator is used to count the number of dogs.

The fourth query involves retrieving all the pets, filtering them to include only those weighing more than 10 kilograms, and then sorting them in ascending order based on their weight.

In the fifth query, the client retrieves all the pets, maps them to obtain their weight, and then uses the ”reduce” operator to accumulate the elements into a single float array. This array stores the sum of the weights, the sum of the squares of the weights, and the number of pets. Finally, another ”map” operation transforms the values in the array into the average and standard deviation.

For the sixth query, the client retrieves all the pets, uses the ”map” operator to transform the birth date of the pets into a string, extracting only the year of birth, and returns a map with the age and name of each pet. The ”reduce” operator is then used to accumulate and reduce the elements of the flux into a single result, comparing the ”oldest” variable to the ”current” element being processed, returning the oldest pet.

In the seventh query, the client retrieves all the pets and groups them by the ID of the owner they are associated with. A ”flatMap” operation is used to obtain the number of pets per owner, and a ”filter” operation excludes owners with only one pet. The counts are collected into a list, and a ”map” calculates the average number of pets per owner.

The eighth query involves retrieving all the owners, using the ”flatMap” operator to obtain the owner's ID and name, and making a new request to retrieve all the pets. The function counts the number of pets with the same owner ID and returns a map with the owner's name and the number of pets. The owners are then listed in descending order based on the number of pets.

In the ninth query, the client retrieves all the owners, uses the ”flatMap” operator to obtain the owner's ID and name, and makes a new request to retrieve all the pets. The function counts the number of pets with the same owner ID and returns a map with the owner's name and the names of the pets they own. The pets are listed in descending order based on their names.

In each query, the results are stored in corresponding files using the ”PrintStream” class. The ”doOnNext” operator is used to insert data into the file, and ”doFinally” is used to close the file.

All the queries are designed to tolerate network failures. The ”retryWhen” operator is used in each query to define the maximum number of retries, which is always three, and the exception that is thrown to the console if those retries are exceeded.

All the retrieved data is converted to a JsonNode for better handling.

The ”ClientApplication” class is responsible for connecting the WebClient to the server URL and invoking all the methods for each query. The ”application.properties” file contains all the necessary configurations, including the client's port.

IV. Conclusion

In this second assigment, we built a web application using WebFlux and reactive programming based on the Spring Boot framework. A reactive Server based on Postgres was created that provides CRUD operations based on web services. A web application was created that utilises the services of the server and enables the management of data about pets and their owners. In a simplified way, the project requests are processed and stored in files on the client side after the queries are made to the server. To make our client application more fault tolerant, the ability to retry all requests up to three times has been added. To understand what is being executed on the server side, logs for the execution of services related to Pet and Owner were added. With this delivery, it was possible to consolidate the content covered in the theoretical and practical lessons on web services and develop a system that integrates three components: Database, Server and Client.

References

[1] Material of practical Integration Systems classes 2023/2024