Report 2

Systems Integration
Spring Reactive

Alexy Almeida	2019192123	adenis@student.dei.uc.pt
Edgar Duarte	2019216077	edgarduarte@student.dei.uc.pt

PL2

November 2022

Contents

1 Introduction

This report aims to show the development of a web application that exposes web services and a client application that will consume those web services. For that purpose, the applications were written using Reactor Flux, meaning there is a Reactive Server, which is a legacy application with basic functionalities and a Client using Reactive code, which collects data from the server and produces several reports.

2 Project structure

The project contains three main applications:

- Main client (client1): runs once and collects data from the server to produce several reports;
- Secondary client (client2): runs once by connecting to the server and populates the database with random data.
- Server (server): keeps running and uses CRUD operations to carry out several functions, which produce the reports client1 inquires.

Besides the applications, three main entities were created:

- Professor: contains id and name;
- Student: contains id, name, birthdate, credits and grade;
- StudentProfessor: contains id, professor id and student id. This entity's purpose serves as a relationship between a student and professor.

The execute specific queries, there are three repositories:

- ProfessorRepository: only contains a findByName query;
- StudentRepository: only contains a findByName query;
- StudentProfessorRepository: contains findByStudentId, findByProfessorId and findByProfessorIdStudentId.

Lastly, to handle the interaction between the users and the server, two controllers were created:

- ProfessorController: contains Post, Get, Put and Delete mappings;
- StudentController: contains Post, Get, Put and Delete mappings;

2.1 Server

The server executes the following queries:

GET getStudents: returns a Flux containing every Student;

GET addRelationship: returns a Mono relationship created between a Professor and a Student;

GET deleteRelationship: deletes a relationship between a Student and a Professor;

GET getStudentProfessors: returns a Flux containing every Professor id for a specific Student;

GET getProfessor: returns a Mono Professor by finding its id;

GET getAllProfessors: returns a Flux containing every Professor;

POST addStudent: adds a Student to the database. Serves as a query for the second client, which populates the database;

POST addProfessor: adds a Professor to the database, in the same context as "addStudent";

POST addRelationship: adds a relationship between a Professor and a Student, serving in the same context as the last two queries.

Besides that, for testing purposes, the server also populates some students and professors, with relationships between them.

2.2 Main Client

The client establishes a connection to the server and automatically retrieves every information at once:

1. Names and birthdates of all students

1. Ask the server to get all the Students in the database;
2. Server returns a Flux with all Students;
3. Use *doOnNext()* to print their name and birthdate.

2. Total number of students

1. Ask the server to get all the Students in the database;
2. Server returns a Flux with all Students;
3. Use *count()* to get the number of Students in the Flux;
4. Use *doOnNext()* to print the value;

3. Total number of students that are active

1. Ask the server to get all the Students in the database;
2. Server returns a Flux with all Students;
3. Use *filter()* to only consider students with less than 180 credits;
4. Use *count()* to get the number of Students in the Flux;
5. Use *doOnNext()* to print the value;

4. Total number of courses completed for all students

1. Ask the server to get all the Students in the database;
2. Server returns a Flux with all Students;
3. Use *map()* to get a Flux with the credits of each Student;
4. Use *reduce()* to sum all the values in the Flux;
5. Use *doOnNext()* to print the values in the Flux divided by 6.

5. Data of students that are in the last year of their graduation

1. Ask the server to get all the Students in the database
2. Server returns a Flux with all Students
3. Save this flux in a variable (*stream*)
4. Use *filter()* to have a Flux with Students that have credits in the interval [120,180].
5. Use *sort()* to sort the Students by their number of credits;
6. Use *doOnNext()* to print the Students information

6. Average and standard deviations of all student grades

1. Ask the server to get all the Students in the database;
2. Server returns a Flux with all Students and saves this flux in a variable (*stream*);
3. Use *map()* on *stream* to get a Flux that containing all the grades, followed by *reduce()* to sum these values. The sum is saved in a Mono variable *sum*;
4. Use *count* on *stream* to get the number of Students and save it to a Mono variable *count*;
5. Use *Flux.zip()* to combine the *sum* and *count* Mono's into a pair value, followed by a *map()* that calls a function *calculateAverage(int, int)* to calculate the mean. Use *doOnNext()* to print out the value. Save the value to a Float *mean*;
6. Use *map()* on *stream* to calculate the standard deviation of each element using the *calculateStandardDeviation(int, mean)* function. Use *reduce()* to sum all these values and follow it with another *map()* that calculates the square root of the sum divided by the size of the flux *count*. Finally, use *doOnNext()* to print the value.

7. Average and standard deviations of students who have finished their graduation

Similar to number 6. The only difference is that *stream* will now have a filter that only considers Students that have 180 credits.

8. The name of the eldest student

1. Ask the server to get all the Students in the database;
2. Server returns a Flux with all Students and saves this flux in a variable (*stream*);
3. Use *reduce()* to get the oldest Student by using the *oldestDate(Student s1,Student s2)* function that returns s1 if s1 is older than s2 and returns s2 if it is older than s1. If they are exactly the same age (including being born in the same day) s2 is returned;
4. Use *doOnNext()* to print the Student.

9. Average number of professors per student

1. Ask the server to get all the Students in the database;
2. Server returns a Flux with all Students and saves this flux in a variable *student_stream*
3. Use *count()* to get the number of Students and save it in a variable *studentQuantity*;
4. Use a *flatMap()* on *student_stream*. In the *flatMap* we use the webSocket to ask the server to return a Flux containing all the Professors for a given Student id. This is done for all the Students in the Flux. The resulting Flux after the *flatMap* will be a Flux of all the Student's Professors. A Professor might appear multiple times in the flux. Finally, we use *count* to count the number of Professors in the Flux and save the value to a variable *studentProfessorsQuantity*;
5. Print the division of *studentProfessorsQuantity* by *studentQuantity*.

10. Name and number of students per professor, sorted in descending order

1. Ask the server to get all the Students in the database;
2. Server returns a Flux with all Students and saves this flux in a variable *student_stream3*;
3. Create a *Map* with the pair (Professor Name (String), Student (Object)) as the (key,value) pair;
4. Ask the server for all the Professors. Using *doOnNext()* creates an entry in the *Map* for each Professor;
5. Use *doOnNext()* on *student_stream3*. In the *doOnNext* the webSocket is used to ask the server for all the Professors ids associated with a specific Student id. A *flatMap()* is used on the resulting Flux so that it can ask the server for all the Professor Objects from the ids given. With the resulting Flux, we add the relationship to the *Map*;
6. A sleep here is necessary because *doOnNext()* does not allow for *block()* or *blockLast()* inside of it. Without this sleep, the Map would remain empty for some milliseconds and the next code would come out empty. It is not the ideal solution but it was validated by our practical classes' teacher;
7. Sort the *Map* by decreasing order based on the Student's array length using *sorted()* to sort the map and *collect()* to create the new map;
8. Use *forEach()* to create a lambda expression that prints the map in order.

11. Complete data of all students, by adding the names of their professors

1. Get all the Students from the server using the websocket.
2. Use *flatMap()* to create a Flux of Strings that has all the information of a Student in them.
3. First, a string is saved in Mono *a*, containing the Students' information except the one of their professors.
4. Get all Professors' ids of the current Student and save them to a variable *teachersId*.
5. Use *flatMap()* on *teachersId* to transform the Professors' ids to Professor Object by using the websocket to ask the server for these objects. After that, a *map()* is used to only obtain a Flux with the Professors' names. Finally, *collectList()* and *flatMap()* are used to verify if the Flux is empty or not. If it is, a Mono saying "No professors" is returned. If it isn't empty, a Mono with all the Professors' names is returned.
6. *Mono.zip* is used to zip *a* with the result from the previous point, to then *map()* them together, forming 1 string for each student.
7. Finally, *doOnNext()* is used to print the full information of each Student.

2.3 Secondary Client

The secondary client, whose main purpose is to populate the database, executes five main functions:

- createRandomStudent: uses Faker library to generate random data for the Student class and returns it;
- createRandomProfessor: same purpose as the previous function, except it's for a Professor class;
- addStudents: returns a list of students with randomly generated data;
- addProfessors: returns a list of professors with randomly generated data;
- addRelationships: creates several relationships between previously created random Students and previously created random Professors;

3 Performance

Some conclusions can be drawn when it comes to the performance of Reactive programming. For that, a table is presented, containing the time (in milliseconds) it takes for the client to run features 9, 10 and 11:

Features			
Amount Elements	Feature 9	Feature 10	Feature 11
10	137	91	80
50	179	151	336
100	321	283	582
500	827	449	1091

Note: For feature 10, the times do not include the *sleep* time.

As can be observed, all features depend linearly on the amount of elements in their Fluxes. It was observed that the usage of Maps with lambda expressions has a better performance than just the usage of Fluxes. On the other hand, this feature uses a *sleep* of around 400 milliseconds, meaning its performance is almost the same as feature 9.

In an initial phase of the development of the application, lambda expressions containing loops were used. The following table presents the time (in milliseconds) that it takes for the client to run feature 7 with and without loops:

Amount Elements	Feature 7 with Lambda expressions	Feature 7 with Flux streams
10	13	33
50	16	55
100	18	75
500	26	114

The main observation made here is that the usage of lambda expressions with for loops is better than only using Flux streams. Even so, for the academic purpose of this project, the Reactive approach is the final solution.

4 Conclusion

Overall, it is considered that the goals proposed for this project were achieved. While, initially, the created queries were using simple operations, such as cycles, we found out that using lambda expressions simplified the process, making the code run in a cleaner way. Besides that, every other conclusion has already been made when observing the performances.