Enterprise Application Integration
Reactor Core/Web Reactive

Carlos Eduardo da Costa Jordão	2019221373
Sofia Yankova	2021230188

1 - Introduction

In this project we were tasked with developing a web application with webservices to that manage media and user. In this report we shall explain how the client performs the requests to the server, and how the requirements work. The main focus of the project is to learn, by creating a client-server application, how React core works and develop skills in this area.

2 - Architecture

The architecture used in this project is designed so that the client has no direct connection to the information stored in the database. To do this we have 3 containers, the database container, the server container and the client container, as seen in figure 1.

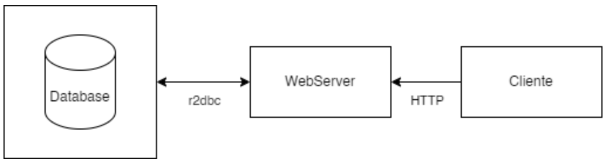


Figure 1 – Architecture Model

Communication done between the server and database is made using R2DBC that provides the server the access to the database, the communication between the server and the client is one way, the client sends the request and simply receives the response of the request, this is done via HTTP.

Since the server is a legacy application the only tasks it must perform are basic CRUD operations to manage the database, the information the client requests is sent in full without any processing being done to the data. The server also has logging which occurs when it receives a client's request.

2.1 - Database Structure

The tables below represent the structure the database has, we store all the user information and media information in the "users" and "media" tables respectively, when we need to add a connection between a user and a media, we simply do it via the "user_media" table that will store the user_id and media_id. The table "user_media" will be a list of multiple instances of the different users and movies, hence why we do an entity relation many-to-one from the "users" and "media" to the "user_media" table.

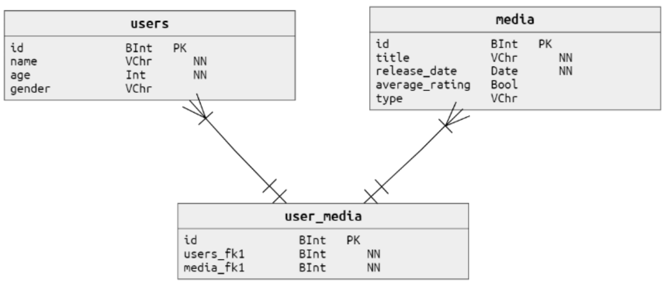
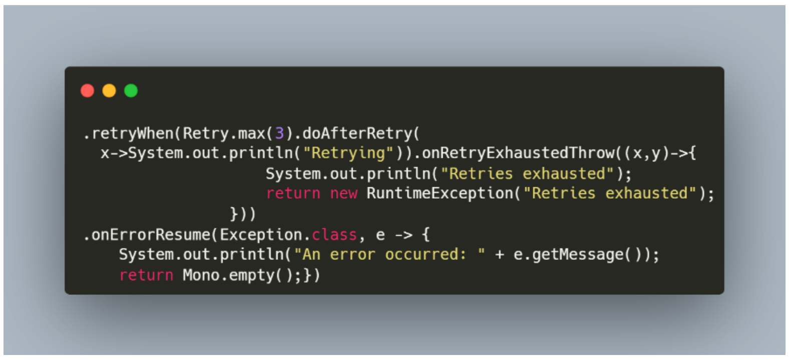


Figure 2 – Database

2.2 - Fault tolerance

We implemented a fault tolerance mechanism in the client side when it performs a request to the server, in the case it fails to connect with the server three times, upon performing the three tries to connect with the server, it will catch the error and stop the execution.



```
.retryWhen(Retry.max(3).doAfterRetry(
  x->System.out.println("Retrying")).onRetryExhaustedThrow((x,y)->{
                  System.out.println("Retries exhausted");
                  return new RuntimeException("Retries exhausted");
              }))
.onErrorResume(Exception.class, e -> {
    System.out.println("An error occurred: " + e.getMessage());
    return Mono.empty();})
```

Figure 2 – Retry & Catch code

3 – Requirements

In this section we shall discuss the methods used to process the data sent by the server for each request, when the information is all parsed and the result is obtained, we write the result in the specific request file. Most of the requirements are straightforward so, although we will explain what we made, we will not delve into it deeper in those cases. In all the requirements we extract the server response as a Flux, meaning it will have multiple instances of the objects that were requested.

For the first requirement we simply obtain all the existing media in the database and with the use of the 'reduce' method to create a string with all the media and the relevant information. In the second requirement we simply request the media information again and use the method 'count' to count the number of media received.

With the third requirement we do the same as before and apply the function 'filter' to filter out the media with a rating bellow 8.0 rating, then we count the media that was filtered using the 'count'.

Unlike the other requirements we needed to access the "user_media" table for in order to obtain the number of media that is subscribed, first we create a HashSet which will be used to store the unique count of a given media, since there will be multiple instances of the same media in the "user_media", to do this we use the 'reduce' where we parse each user_media and store the first time a new media appears in the HashSet, then we simply use 'count' to get the number of media that are subscribed.

In the fifth requirement we obtain all the media and apply the 'filter' function to get the media that is dated in the 80's, then we use the 'sort' to compare the average rating and make them ordered in descending order.

On the sixth requirement, after obtaining all the media we transform each Media object into their respective average ratings (since the other media information is irrelevant) then in the 'reduce' do the sum of all the rating, the count of ratings and the sum of squared ratings (R^2). Then we do the calculations in the 'map' and store in a double array.

For the seventh requirement we obtain the media, then we use 'reduce' to compare the release dates between the media. If a media is older than the current one it switches, otherwise it maintains the same one.

In the eighth requirement we first obtain the "user_media" and "media" information, then using 'flatmap' (not 'map' so we don't have a multiple Flux's) and using the 'filter' we obtain the number of users for each media that is subscribed. Then in the 'reduce' we sum all the users in each user_media that was counted before, after we use 'zipWith' to count all the media in 'mediaFlux' and combine with the sum of the users we did in the reduce. And finally, we perform the calculation for the average users per media with the values we obtained.

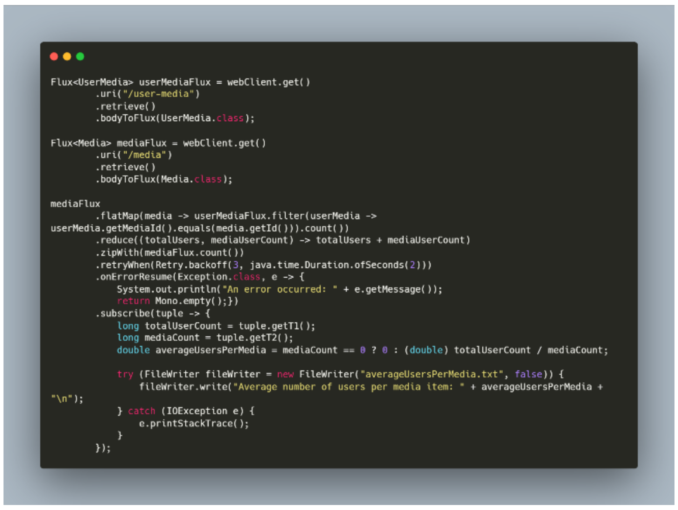
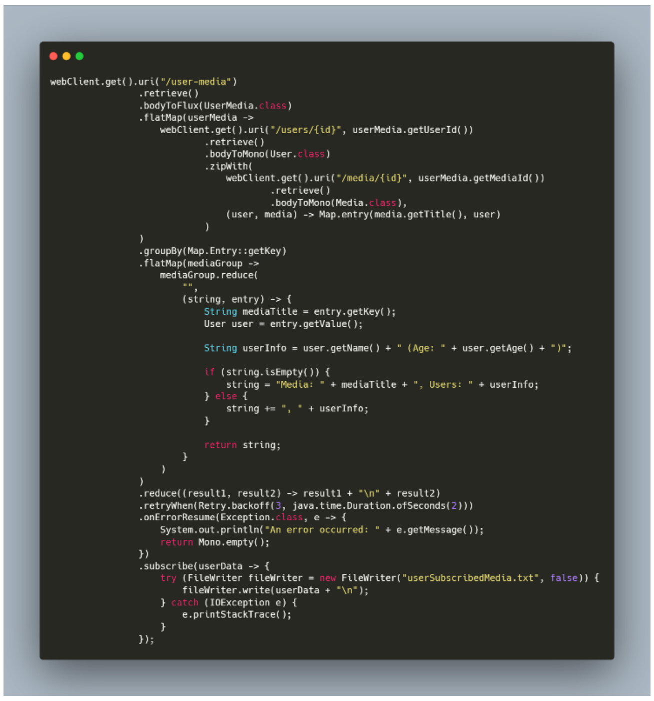

```java
Flux<UserMedia> userMediaFlux = webClient.get()
        .uri("/user-media")
        .retrieve()
        .bodyToFlux(UserMedia.class);

Flux<Media> mediaFlux = webClient.get()
        .uri("/media")
        .retrieve()
        .bodyToFlux(Media.class);

mediaFlux
        .flatMap(media -> userMediaFlux.filter(userMedia ->
userMedia.getMediaId().equals(media.getId())).count())
        .reduce((totalUsers, mediaUserCount) -> totalUsers + mediaUserCount)
        .zipWith(mediaFlux.count())
        .retryWhen(Retry.backoff(3, java.time.Duration.ofSeconds(2)))
        .onErrorResume(Exception.class, e -> {
            System.out.println("An error occurred: " + e.getMessage());
            return Mono.empty();})
        .subscribe(tuple -> {
            long totalUserCount = tuple.getT1();
            long mediaCount = tuple.getT2();
            double averageUsersPerMedia = mediaCount == 0 ? 0 : (double) totalUserCount / mediaCount;

            try (FileWriter fileWriter = new FileWriter("averageUsersPerMedia.txt", false)) {
                fileWriter.write("Average number of users per media item: " + averageUsersPerMedia +
"\n");
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
```

Figure 3 – Requirement 8

For the ninth requirement, we start by obtaining the “user_media” data, then using ‘flatmap’ we fetch the user and media details and combine them with ‘zipWith’ to create a Map with the media title being the key and the value the user. After we use ‘groupBy’ to combine them by the media title, getting a media title key for each group and the map we created with ‘zipWith’ inside containing the media title and associated users, we then reduce to join the users into a single string.

And finally, we use ‘reduce’ to combine all the results of the media titles and associated users into a single sentence.



```
webClient.get().uri("/user-media")
                .retrieve()
                .bodyToFlux(UserMedia.class)
                .flatMap(userMedia ->
                    webClient.get().uri("/users/{id}", userMedia.getUserId())
                            .retrieve()
                            .bodyToMono(User.class)
                            .zipWith(
                                webClient.get().uri("/media/{id}", userMedia.getMediaId())
                                        .retrieve()
                                        .bodyToMono(Media.class),
                                (user, media) -> Map.entry(media.getTitle(), user)
                            )
                )
                .groupBy(Map.Entry::getKey)
                .flatMap(mediaGroup ->
                    mediaGroup.reduce(
                        "",
                        (string, entry) -> {
                            String mediaTitle = entry.getKey();
                            User user = entry.getValue();

                            String userInfo = user.getName() + " (Age: " + user.getAge() + ")";

                            if (string.isEmpty()) {
                                string = "Media: " + mediaTitle + ", Users: " + userInfo;
                            } else {
                                string += ", " + userInfo;
                            }

                            return string;
                        }
                    )
                )
                .reduce((result1, result2) -> result1 + "\n" + result2)
                .retryWhen(Retry.backoff(3, java.time.Duration.ofSeconds(2)))
                .onErrorResume(Exception.class, e -> {
                    System.out.println("An error occurred: " + e.getMessage());
                    return Mono.empty();
                })
                .subscribe(userData -> {
                    try (FileWriter fileWriter = new FileWriter("userSubscribedMedia.txt", false)) {
                        fileWriter.write(userData + "\n");
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                });
```

Figure 4 – Requirement 9

4 – Conclusion

In conclusion, on the completion of this project we gained a better understanding of how reactive applications work and the difficulties it comes with, by learning how to manipulate flux data and implementing more efficient ways for the client to perform the request with the server.