Project - Phase 8 Report

Group 14
Tiago Carvalho fc51034
Diogo Lopes fc51058
Miguel Saldanha fc51072
João Roque fc51080
João Afonso fc51111

16/05/2021

1 Motivation

The idea for our project emerged when we were wondering how cool it would be to have an API that could help us decide which shows to watch next based on our personal taste. We figured that such a service could be developed with relative ease if users marked shows as viewed and/or liked.

Because users can watch more than just movies, for example animes, we wanted to use more than one dataset. Both animes and movies can have a lot in common not only with each other but also with books, so we also decided to use a book dataset. Using data from multiple datasets would give us a more realistic experience when it comes to cloud-native applications development, since these applications use data from so many sources.

With three datasets, we aimed to be able to effortlessly search through any of them. We wanted users to be able to mark movies, animes or books as seen or liked, and get suggestions of what to see next. Since the datasets have very similar categories among them, suggesting books mixed with movies and animes would be a possibility that we thought would add value to our application.

Our idea to implement the suggestion mechanism was to base these suggestions on the user's likes and views, which would indicate to us which categories the user prefers, and, therefore, allow us to suggest good movies, animes or books to the user.

We called our API "Seen", since users can see movies, animes and books and then get suggestions based on their profile, on what they have seen.

2 Dataset characterization

2.1 Dataset 1 — IMDB

This dataset provides a lot of information about movies and shows that can be seen in IMDB.

We downloaded the dataset (updated one year ago) from the Kaggle website.

From the whole dataset, these are the columns that were important to us:

Columns	Example
id	606e2683b3fff1da8a207ae9
name	The Arrival of a Train
category	[Action,Documentary,Short]
rating	7.4
type	short

Table 1: Movie example in our database

2.2 Dataset 2 — MyAnimeList

The second dataset, regarding the MyAnimeList website, was obtained from Kaggle.

This dataset not only has a lot of anime content, but also user information, but because we want to connect with the other datasets it doesn't make sense to use that data. We used the following columns:

Columns	Example
id	606e252aebddc73ebfb15507
name	Shakugan no Shana: Season II
category	[Action,Drama,Fantasy,Romance,School,Supernatural]
rating	7.72
imageUrl	https://myanimelist.cdn-dena.com/images/anime/10/18669.jpg

Table 2: Anime example in our database

2.3 Dataset 3 — GoodReads

At last, this dataset represents books from the GoodReads website, also downloaded from Kaggle.

The columns that are meaningful for us to be able to use this dataset together with the animes and movies datasets are the following:

Columns	Example
id	606e25ad5e927a606f534284
name	Of Mice and Men
description	The compelling story of two outsiders [...]
category	[Classics,Fiction,Academic,School,Literature,Historical]
rating	7.7
imageUrl	https://images.gr-assets.com/books/1511302904l/890.jpg

Table 3: Book example in our database

3 Use cases

We have 3 types of Users: an Admin, which is a logged-in user with special permissions, a Regular user, which is a logged-in user, and a not logged-in user that we call Any.

Services	User	Functionalities
Normal	Any	Sign in See Book, Show and Movie Library
	Regular	User Log in Set Book/Show/Movie as seen Set Book/Show/Movie as liked Ask for suggestions to read and/or watch Count how many views a specific Item has Count how many likes a specific Item has Top 10 Items with more likes
	Admin	Add Book/Show/Movie to Library Remove Book/Show/Movie from Library
Spark	Any	See best Director and his movies with cast See which Actor has the most connections

Table 4: Use cases

4 API

User	Path				get	post	put	del	description
Regular	/lib	/{page}			×				Returns a *page* from the database
	/suggest					×			List of suggestions to watch
Admin	/item					×			Creates an item to add to the database
Any	/item	/{type}	/{id}		×			×	Gets/Deletes item with specific *id* and *type*
Regular	/item	/{type}	/{id}	/seen			×		Marks item as seen
	/item	/{type}	/{id}	/like			×		Marks item as liked
Any	/item	/{type}	/{id}	/views	×				Returns Item's number of views
	/item	/{type}	/{id}	/likes	×				Returns Item's number of likes
	/getTopTen	/{type}			×				Returns top ten most liked Items with *type*
	/user					×			Creates User
	/user	/login			×				Logs in
Regular	/user	/logout			×				Logs out
	/user	/search	/{username}		×			×	Searches/Deletes User by username
Any	/{director}				×				Returns list with the best Director's movies and his cast
	/actor				×				Returns the Actor's name with movies with the biggest cast in total

5 Architecture (application and technical)

5.1 Diagram

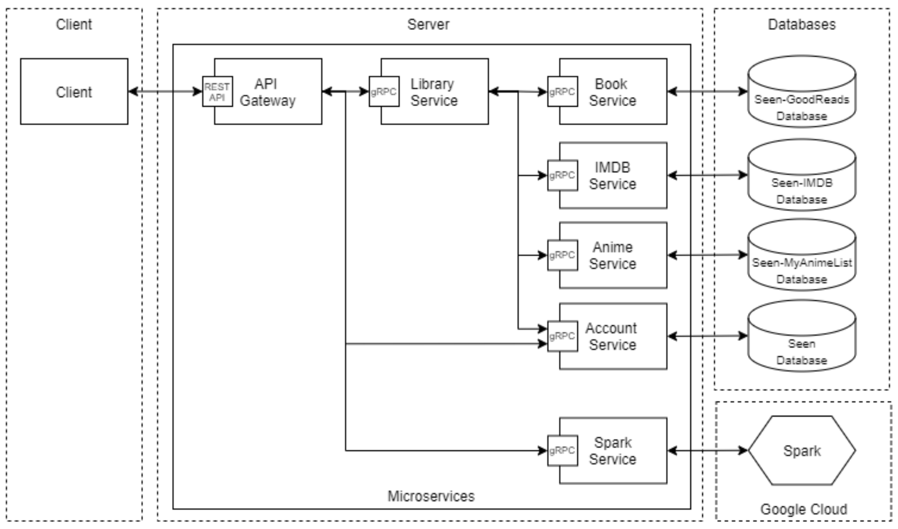

Figure 1: Project's architecture.

5.2 Application

5.2.1 Client

The Client should be able to access our API on his browser:

`https://recommendations.sytes.net`

The Swagger provides a user interface to use and test our calls by adding "/ui" to the end of the url above.

5.2.2 Server

In total there are 7 different microservices working at the same time. Every single one of them runs on Google Cloud, inside the same cluster but in different dockers.

Our reasoning was to have an entrance microservice, which would redirect the request to the microservice responsible for that type of request, for example when sending a request for a page in our library, the API Gateway receives that request and sends it to the Library Service, which is responsible for asking for books, movies and animes to the Book, IMDB and Anime Services, respectively, and then put them together in just one response, which is then sent to the API Gateway, to be shown to the Client.

This API Gateway service also has the responsibility of transforming the REST requests from the Client to gRPC requests that are used internally, between Services.

From the total of 7 microservices, 5 are responsible for the database connection, meaning that they are responsible for translating the request they receive into inserts, updates, removes or queries to the database. They are also responsible for translating responses from the database into responses that can be understood by the other microservices.

5.2.3 Databases

Every database has a service that has the responsibility to access and manage it. While 3 of the databases are hosted by MongoDB, a NoSQL database, the last one is an SQL database hosted on an external server. This last database was initially hosted on Google Cloud, however we removed it from there, because it was costing us a lot of money.

For the Books', Movies' and Animes' databases, we used a NoSQL database since we might have had to change the format of our documents, meaning that if we had an SQL database we would need to drop the entire database and repopulate it again every time we decided to change the schema. MongoDB provides a very easy and intuitive python implementation to work with, which was also a factor to consider when selecting the type of databases that we would be using.

However, for the Users' database we used an SQL one, because we already knew what we wanted from the User and we knew we would use structured data for it.

5.2.4 Spark

Later in the course of the project, we added the Spark microservice. This service receives the gRPC requests from the API Gateway service and then processes them, creating a job to send to Google Cloud where we have a Cluster with the sole purpose of running this type of jobs.

5.3 Technical

5.3.1 Server (Microservices)

TODO

5.3.2 Databases

TODO

5.3.3 Spark

TODO

6 Implementation

We actually made 2 different implementations, one that runs locally and other that runs in the cloud. We created a local version of our application for several reasons: we were more familiar with the development of local-running applications, running the application on the cloud costs money, and we can debug the app better when it is running locally which is important during development. When developing a feature, first we would put everything running locally, by running a script, and once the feature was running without issues on our machines, we would adapt it to run on the cloud.

6.1 Server (Microservices)

6.1.1 Docker Compose

For the microservices deployment we use a docker-compose file, which defines each microservice. In this file, we define a microservice's name, a build stating that we are using a Dockerfile and where the context of the microservice is (its folder, with its required content). We also define the image, hostname and container for the microservice, for which we use the same name as the microservice's. Finally, we assign a port to the microservice, which is crucial for the application to behave properly.

We do the steps stated before for every single microservice inside the docker-compose, but that's not all we have to do. Since the API Gateway service sends requests to the Spark Connector, Account and Library services, we add them as environment variables, meaning that when we run the python file defining the business logic of the service we can fetch the generated IP and then use it to connect to the services with gRPC channels. For the library we add all the services responsible for the databases, which are the Book, Imdb, Anime and Account services.

6.1.2 Dockers

To create a docker image for each microservice to be containerized we used dockerfiles. These files are really simple, they basically have everything they need in the folder they are in, and just make a new folder called service, and copy the current folder inside the newly created one. After copying, they run the installation of the requirements, using pip and the "requirements.txt" file. Then, the same port used on the docker-compose file before, is exposed to be accessible from outside this controller ambient/sandbox. Finally, the entry point is defined as the python file containing all the business logic of the microservice.

6.1.3 Protobuf Files

The protobuf files we created are the basis of our services communication between each other like we stated before.

This files can be seen like interfaces that the services need to comply, so the recipient of the message processes it.

We created an "utils" protobuf file, this file stores the interface of two messages, Success and Empty, and since they are repeatedly used we have them here to be ready to be imported. This file also stores an enumerated, with every Item type supported on this API: BOOK, SHOW and ANIME.

Then the protobuf files are very straight forward, we define a protobuf service with the name of the actual Service that receives requests, and inside that protobuf service the functions that this Service supports.

Those functions are also very simple, they consist in declaring a function that the Service has inside, with a message input and output, and the Service just has to comply with those restrictions, meaning that he receives a message of some type, and needs to "unpack" it and then "pack" it again to return as a response.

Because we import the "utils" file, we then can use the Type (BOOK, SHOW or ANIME) that we declared inside other messages declarations. And use the "repeated" as a crucial part of our messages, since we usually want to send lists back and forth with our Item's content.

To build the protobuf files into importable python files with all the necessary functions for communication we use a bash script, which compiles them and copies each one to the Services that use them.

6.2 Databases

Because we are using the free subscription of MongoBD, we can only have up to 512 Mb in each Cluster we create, this means we cannot fit inside one cluster all the data we have to store.

To resolve this problem we created 4 different mongodb clusters, one for each Item type, and an extra one for the movie database, since the movie data set takes roughly 1Gb we had to split it in 2 halves.

To upload our datasets, we used a script. The script connects to the database, then reads 100000 lines from the dataset, adapts each one with the relevant data to upload and then sends those 100000 items to the database.

Those adaptations are simple, using the csv package we can read both csv and tsv files row by row already split by columns. Then we transform the categories string into a list of strings, separating each category, facilitating queries. At last, we also strip the string from the white space at the beginning and at the end.

About key duplicates we could have that problem using different databases, the generated key in one database could be the same as another one in another database, even if not likely it could happen. To resolve that problem we always use the type of the Item, with that type we choose which Service to send the request to, identifying the id and avoiding the "deletion" by omission of Items, because the first id found would be the one returned, omitting other results.

For the movie database we also prepend (insert at the beginning) a database identification ("1" for the first half of items and "2" for the second one).

For the user database we used another strategy and started an SQL database using SQLAlchemy to connect.

To use a secure connection between the Service to handle the users' data we initialize a connection with certificates (SSL server certificate authority, SSL server certificate and an SSL client key).

To maintain the same structure inside and outside the database we also created a python file named models, which has the correspondent classes to the tables inside the database, this way we can use those classes to fetch results already formatted.

6.3 Spark

7 Evaluation and validation

To perform an evaluation on the system by load testing, we used three different technologies: Prometheus, Grafana and Locust.

Prometheus is a monitoring system that retrieves data directly from the cluster as it is executing, while Grafana displays and amalgamates the data in a way that can be analyzed by the human eye (plots, etc.).

To perform the load tests on itself, we used Locust, a load testing tool.

All these tools were configured automatically with scripts to further automate the deployment of the system (see locust.sh and add_plugins.sh).

7.1 Evaluation

We ran Locust, the tool to perform the load testing, in another cluster to avoid affecting the results. It was tested with up to 800 users, which at that point it was getting close to 100% CPU usage, according to Grafana.

At the time, we were using weak machines (12 CPUs) as Google Cloud's free trial did not allow any higher than that. Every aspect of the system was being tested: getting entries, putting new entries, get pages from the database, get a recommendation (and even the login and logout which was later changed and so there is no need for load testing as it is handled by an exterior service, auth0).

We did load testing with 100 users, 500 users and 800 users which we assume to be a somewhat comfortable limit for the resources we have, it can go higher, but it could probably start running into performance problems after a certain point.

7.2 Validation

TODO

8 Cost analysis

TODO

9 Discussion

9.1 Results

9.2 Analysis

10 Conclusions

10.1 Contributions

10.1.1 51034 - Tiago Carvalho

10.1.2 51058 - Diogo Lopes

10.1.3 51072 - Miguel Saldanha

10.1.4 51080 - João Roque

10.1.5 51111 - João Afonso

10.2 Future alterations

11 Appendix