Technical Report - Project specifications

2SeeOrNot

Course:	IES - Introdução à Engenharia de Software
Date:	Aveiro, 02-12-2020
Students:	93147: David Morais 92964: Mariana Ladeiro 89318: Pedro Iglésias 93442: Wei Ye
Project abstract:	Web app that provides a fast and enjoyable way of browsing and acquiring tickets. The goal is to develop an app that lists available movies and cinemas to the user, so that they can check movies plot and information, as well as schedules and be able to buy tickets for various movie sessions. In addition, it also provides a clean and user-friendly interface that allows users to manage their bought tickets via a developed API. Furthermore a cinema owner can also create a page for its own cinema.

Table of contents:

1 Introduction

2 Product concept
- Vision statement
- Personas
- Main scenarios

3 Architecture notebook
- Key requirements and constrains
- Architectural view
- Module interactions

4 Information perspective

5 References and resources

1 Introduction

This project is going to develop on the scope of IES, and the main objective is:

- ➔ Development of the product specification, from use cases to technical design.
- ➔ Propose, justify and implement a software architecture based on business structures.

2SeeOrNot is a web app to allow and facilitate the marketing of movie tickets.

We created a git repository to develop our product and PivotalTracker backlog so that we could create and track our objectives.

2 Product concept

Vision statement

Our product is based on showing detailed information about each movie.

So, it is possible for each unregistered user to:

- Search for a movie;
- Search for popular movies;
- Search for recent movies;
- Check movie info (it will not show current ticket sales stats);
- Search for cinemas;
- Check premiers;
- Check schedules;
- See comments;

When registered we have a normal user that can:

- Do the same as an unregistered user;
- Can add a movie as favourite;
- Can buy tickets;
- Change profile options;
- Get his favourite movies;
- Get his favourite cinemas;
- Check notifications;
- Check payments;
- Remove movies or cinemas from favourites;

Also, each cinema will be another type of user that will be able to:

- Do the same as an unregistered user;
- Can change profile options;
- Create premiers;
- Create rooms;
- Create schedules;
- Check current rooms;
- Delete schedules or premiers;

There is also an admin user type that will be able to:

- Accept cinema users;
- Add movies;

This product will facilitate the life of the users that will join a large number of movies and cinemas so that it can be easier to buy and sell tickets.

Personas

Marta

Marta is the owner of an independent cinema in downtown Lisbon. Due to the business being slow she wants to add visibility to the business without losing the independent soul of the cinema. So, she's looking for a platform where she can add her cinema and sell tickets with all the gains being reverted to her and her staff.

Lídia

Lídia is 75 years old and lives in Aveiro. Because she is retired, she has a lot of free time and likes to spend it at the movies, especially with her grandchildren. Since Lidia isn't very familiar with technologies, she is looking to find an app with a simple and easy interface.

Samuel

Samuel is a 32 year old engineer living in Coimbra. Samuel is currently working on a mobile app that requires the use of an API with movie details. He's looking for a simple, perceptible and well documented API in order to make his job easier.

Main scenarios

First Scenario

Marta decides to use the app/website to add her independent cinema. To do this, she fills out a form giving the cinema information and submits it. The request is sent to the admin and, if accepted, she can choose the movies playing in her cinema every week based on the movies available as well as schedules, room and seat info.

Second Scenario

Lídia's grandchildren decided at the last minute they would like to go to the movies. Lídia opens the website to see some animation movie options. The movie they decide to watch is almost starting, so, to make the process quicker, Lídias uses the website to purchase the tickets and get in time to the cinema. After the movie, Lídia and her grandchildren give it a rating.

Third Scenario

Samuel is in charge of developing an app that shows movie details, like actors, production and plot. He's now taking care of passing the data from an API to the app. While researching, he finds the 2SeeOrNot website API and decides to use it. In order to do this, he uses the method GET from the API, receiving the data he wanted in JSON format.

3 Architecture notebook

For this project, we used Frontend Backend Separation Architecture. Backend and frontend components were developed separately.

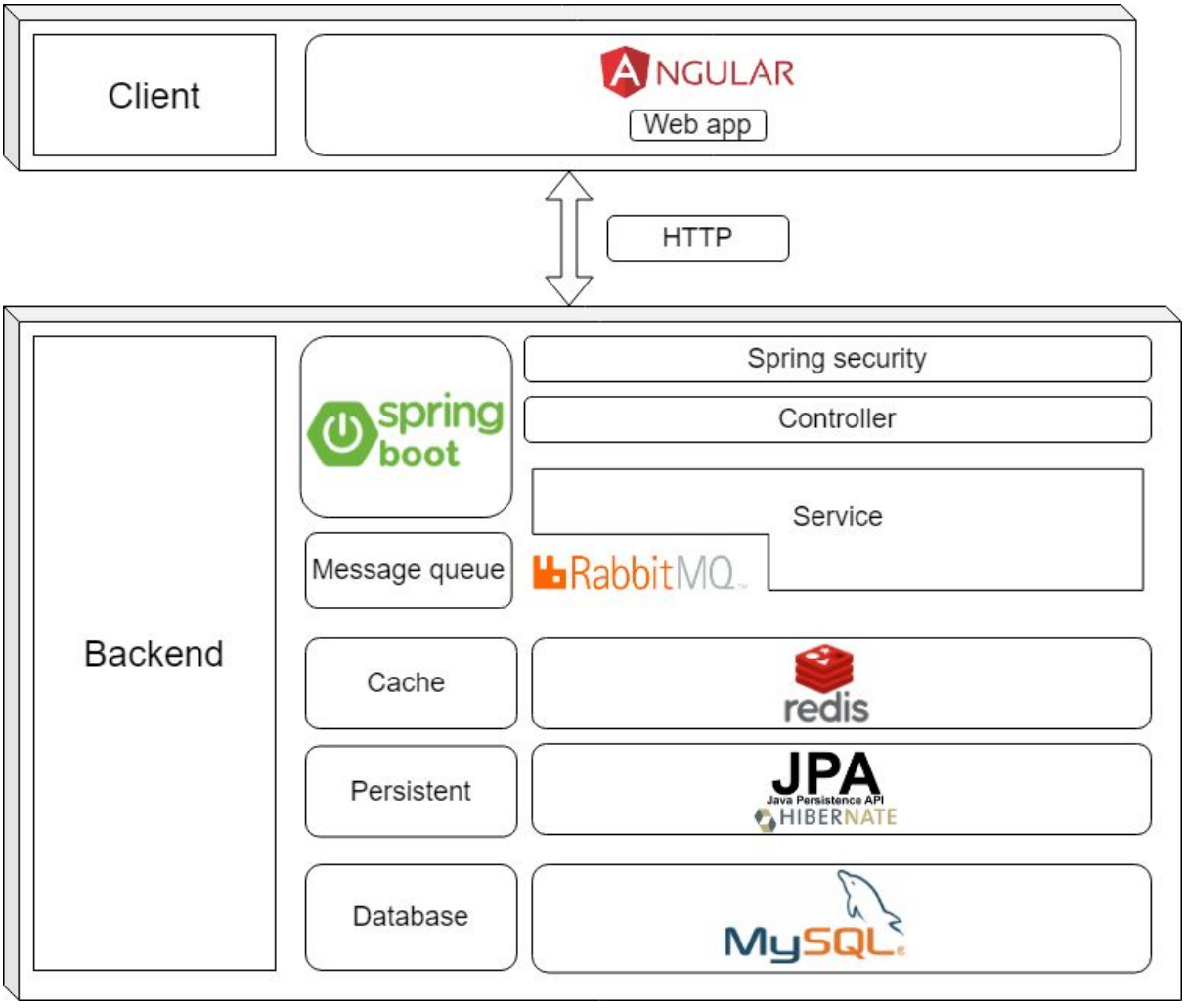
Key requirements and limitations

- The system must provide a user authentication system.
- Different types of users should have different API access. If a user doesn't have access, the request should be refused.
- The system should avoid, as much as possible, the access to the database (use the cache engine).
- The system should guarantee data consistency (example: failure while buying tickets can't cause data inconsistency).
- The system should implement every use case.
- The server should communicate with the frontend through the http protocol.
- The system should analyse the acquired data.
- The system should use a message queue to optimize a few time-consuming operations.

Architectural view

- **Technologies**
  - **Database**
    - Mysql: It's one of the most common relational dbs on the market that we are more familiarized with. It serves to persist the main data of the app.
    - Redis: A NoSql db of the type Key-Value, extremely efficient, provides more data structures than other Key-Value dbs (ex: memcache). The cost of accessing a db (mysql) is expensive, we should avoid it, as much as possible, so we used Redis as a cache of the db queries results.
  - **Backend**
    - Springboot

- **Message Queues**
  - RabbitMQ. In some services we need to send emails to customers, but the sending can take some time, so a message queue is essential. When we need to send an email, we create a specific message, and put it in the rabbitmq queue, when the server is not busy, it will consume the message, and send the email to the client.
- **Authentication**
  - Spring Security + JWT. Used for the user authentication and authorization of the received requests (we have to verify if the user that sends the request has access to a certain API or not).
- **Frontend**
  - Angular



we separate the frontend and backend into two different components. In the client component, we have our web app, which essentially shows clients the data, is programmed in Angular.

The communication between the two components is through the http protocol. All data returned by the server, will be in json format.

In the backend component, we have several layers. The first layer, the spring boot, is responsible for order retrieval, verification of authentication and client authorization and data processing, etc. When a client sends a request to the server, the request will enter the filters of spring security, where we check the authentication and authorization,

As our system has 3 different types of clients, normal client, cinema, and admin, this process of checks is essential, if it does not give any error, the request will get to our endpoints. And finally, the endpoints will call up appropriate services to process the client's request.

Usually, the services process the data, and turn it into a suitable format.

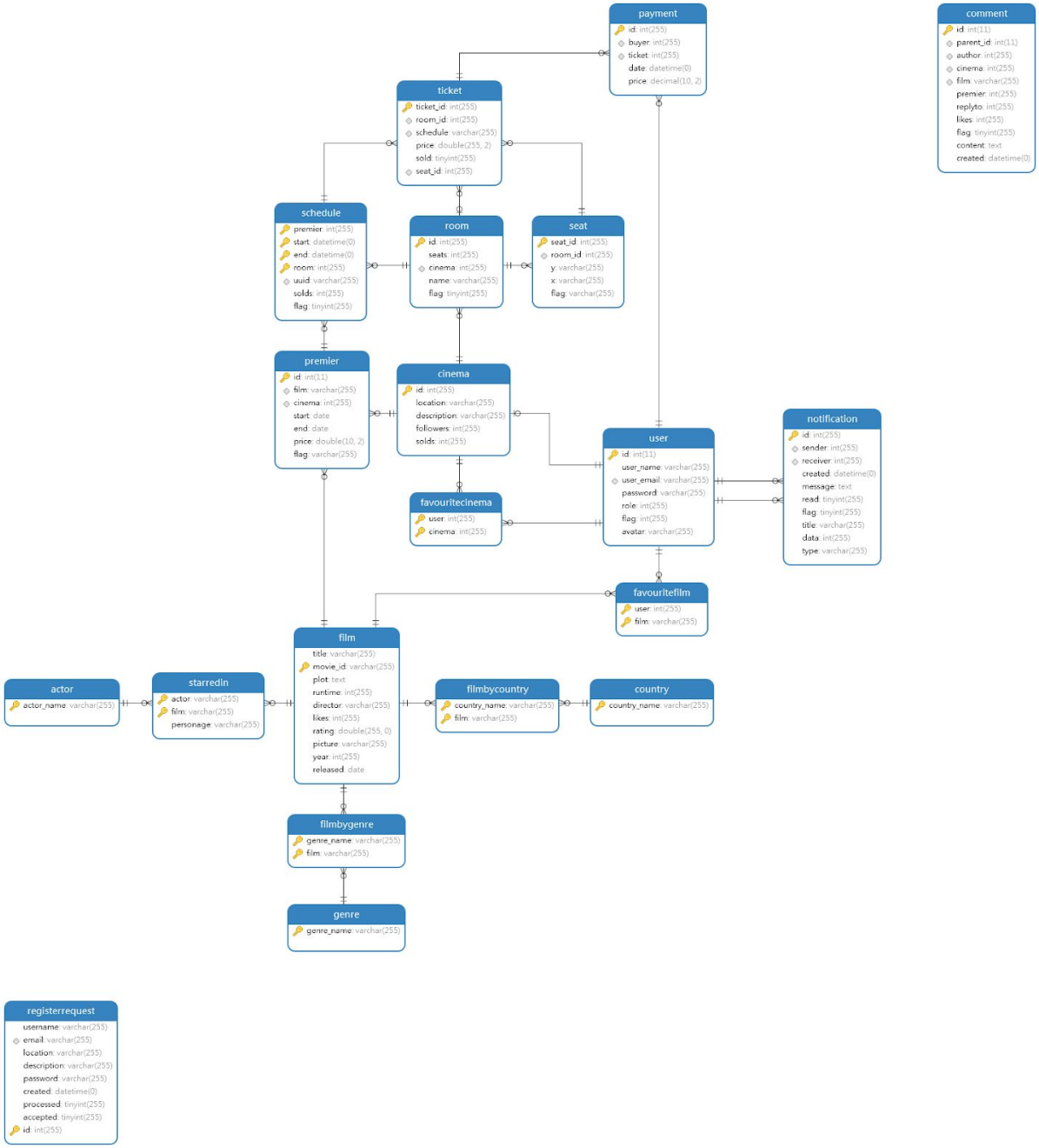
In this case, we have a special service -- mailService, which is used to send mails to customers. But sending emails can take time, so we integrate rabbitmq to optimize the sending process. Basically, when we need to send an email, we put a specific message in a rabbitmq queue and immediately return a "success" message, then, when the server can consume the message, it will call the mailservice method.

The next layer is Cache, as the cost of accessing the database is expensive, we should avoid it whenever possible. When a service wants to query the database, it must first check if the data it wants to search is available on the network or not, if so, it reads the data and returns, otherwise it accesses mysql.

The persistence layer, there is nothing special, we use the hibernate JPA to interact with mysql.

And finally, the last layer, our database, mysql.

- **Database model**



User	Table that stores the system users. A user can be a normal client, a cinema owner or the admin.
Cinema	Table that stores all the cinemas, each cinema is also a user.
Film	Table that stores all the information about each movie.
FavouriteFilm	Table that stores each user's favourite films.
FavouriteCinema	Table that stores each user's favourite cinemas.
Country	Table that stores all the countries (of films).
Comment	Table that stores all comments generated by clients
Payment	Table that stores all the ticket purchases made.
Ticket	Table that stores all the tickets.
Genre	Table that stores film genres.
FilmByGenre	Table relating the film to one or more genres.
FilmByCountry	Table relating the film to one or more countries.
Notification	Table that stores all the notifications sent from a cinema to a client.
Actor	Table that stores all the actors.
StarredIn	Table that relates the actor with the film.
Premier	Table that stores all the premieres.
Schedule	Table that stores all the schedules.
Room	Table that stores all the rooms of each cinema.
Seat	Table that stores all the seats of each room.
RegisterRequest	Table that stores all the register requests of cinema.

Module interactions

- **Frontend communication**
  - The communication is made through HTTP. The frontend sends a http request to the server, this request can be rejected due to authentication/ authorization failure. If not, it enters one of the backend endpoints. AFter processing the request, the backend returns a serialized JSON object (that contains the response, http status code and other necessary information) to the frontend.

- **Json Protocol**

  response: {

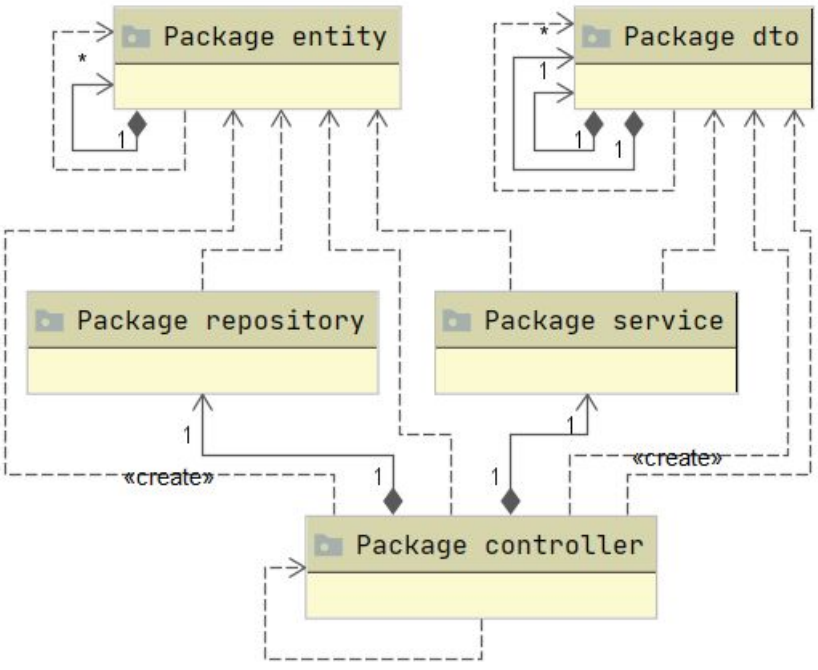
  “statusCode”: xxx (http status code defined by us),

  “message”: xxx (message that describes an answer, ex:success, authentication failed, etc.),

  “data”: xxx (data returned from the server)

  }

4 Information perspective



We have a lot of classes, we can't show them all in a UML diagram. So we decided to show UML between packages. With this diagram, it is a little difficult to understand the dependency between the packages, it seems that everything depends on everything, but in reality there is a sequence of the invocation very explicit: Controller -> Service -> repository. The controllers receive the http requests, invoke service methods, and the services consult the database, manipulate the read data, encapsulate it in the DTOs (Data Transfer Object).

Finally, controllers receive the data returned by the services, put them in a Result Class object (encapsulates the httpStatusCode, the message and the data, used to communicate with the frontend), serialize them in json format, and return them to the frontend.

Data generation

Our data generation is coming from a python script that loads a dataset form the imdb website and loads each imdb id that is available and is valid. This “is valid” means that the movie is recent released in the current year / last year.

5 References and resources

Used for data generation:

-https://www.imdb.com/interfaces/

-https://imdbpy.github.io/