DOCUMENTATION

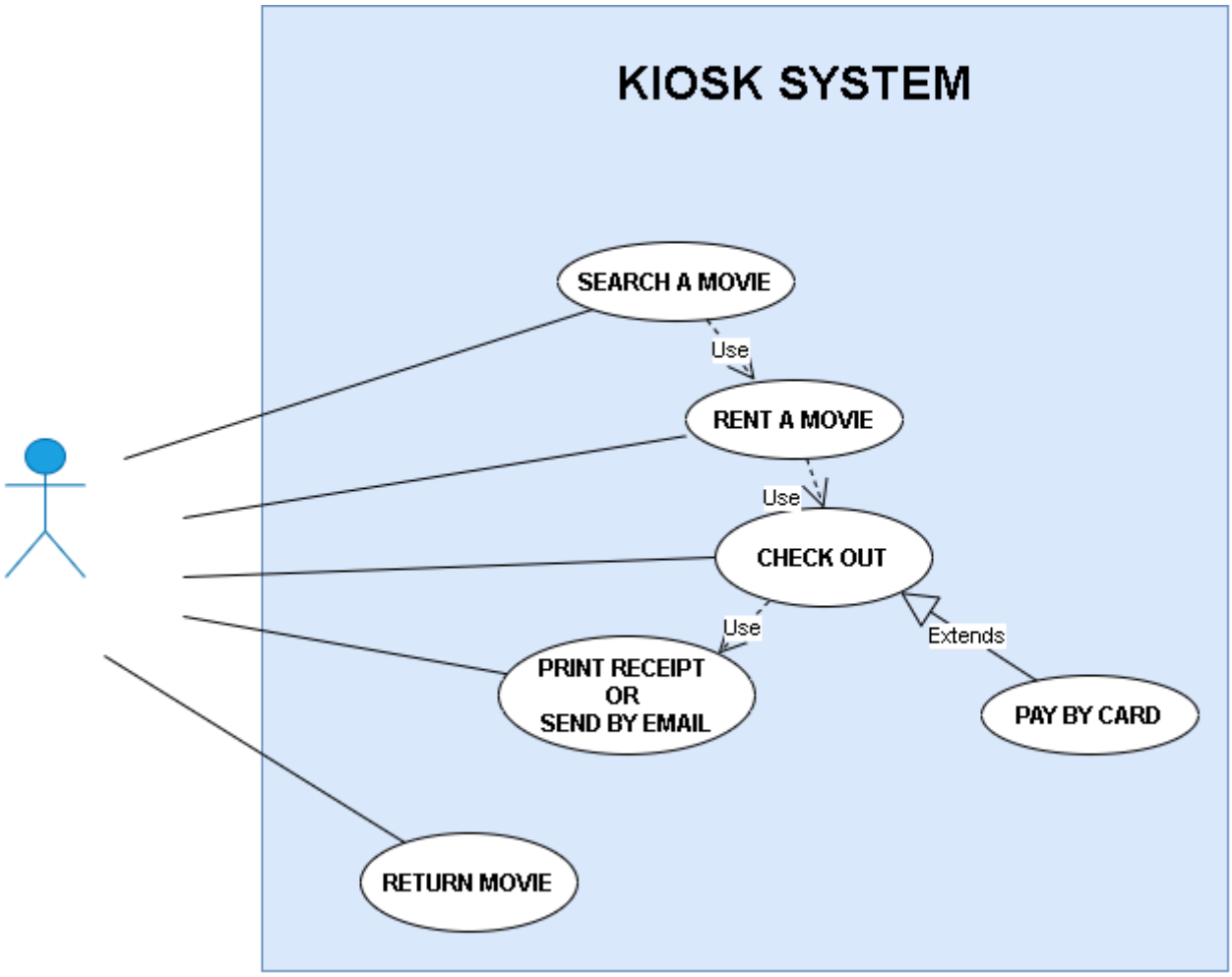
Henrique Demetrio Pacheco de Souza || Jose Paulo

CCT COLLEGE 2022

Contents

VIEW OF KIOSK SYSTEM

1. USE CASE DIAGRAM



1.1. SUCCES SCENARIO

Actor	Client
StakeHolder	Xtra-Vision
OverView	This use case consists of the movies selection process

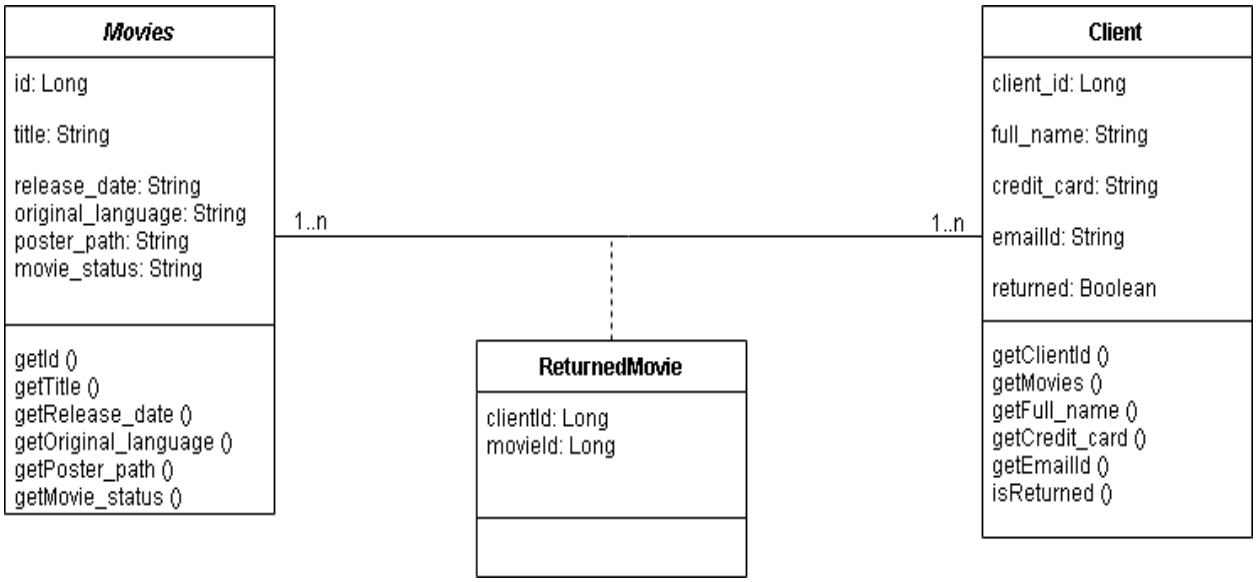
Main Success Scenario
1. The **system** makes available the entire catalog of movies. 2. The **customer** selects the desired movie. 3. The **system** provides the customer with more details about the movie. 4. The **customer** adds the film to the cart. 5. The **system** informs the shopping cart with a brief summary of the order. 6. The **customer** wants to make the purchase. 7. The **system** displays the payment methods. 8. The **customer** chooses the payment method 9. The **system** informs you of the order confirmation.

1.1. UNSUCCESSFUL SCENARIO

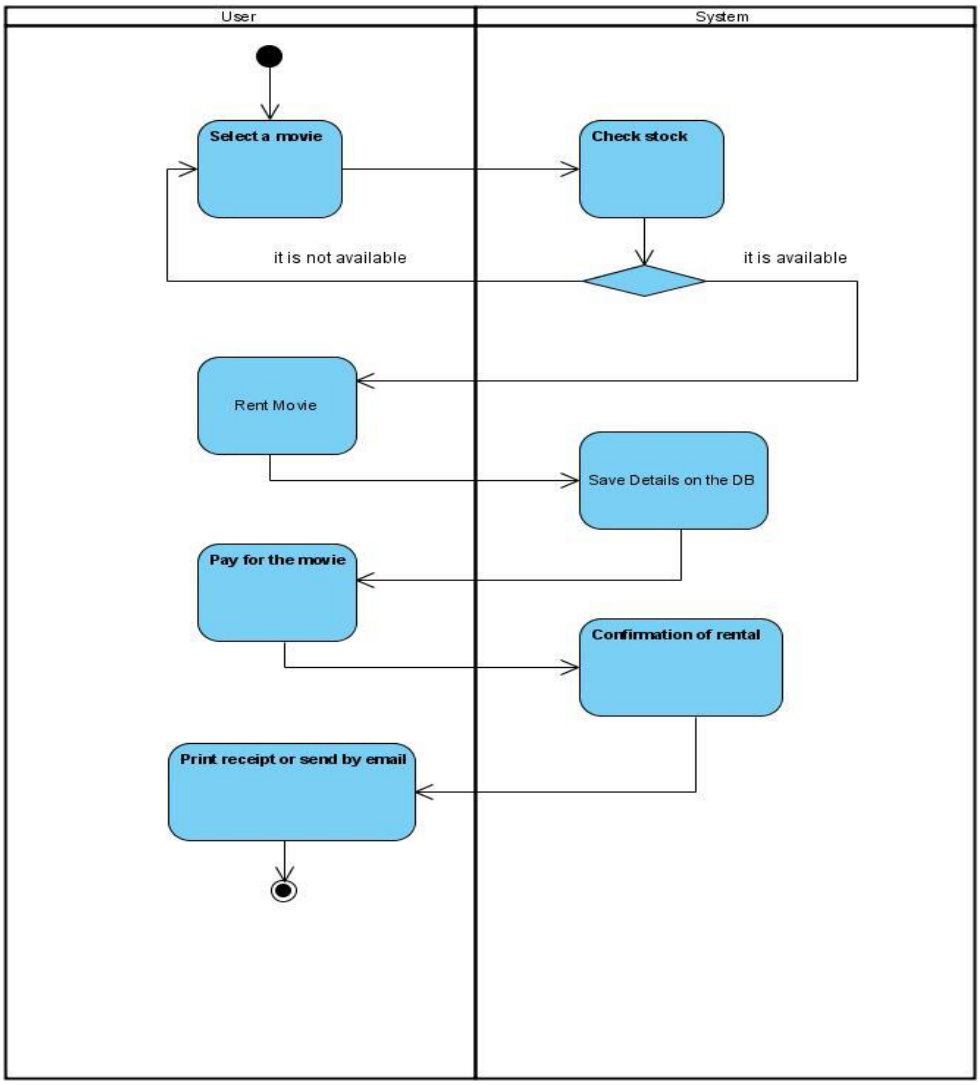
Unsuccessful Main Scenario
1. The **system** makes available the entire catalog of films.

2. The **customer** selects the desired film.
3. The **system** provides the customer with more details about the film.
4. The **customer** adds the film to the cart.
5. The **system** informs the shopping cart with a brief summary of the order.
6. The **customer** wants to make the purchase.
7. The **system** displays the payment methods.
8. The **customer** chooses the payment method
9. The **system** informs that something has gone wrong with the details provided, and ask for confirmation.

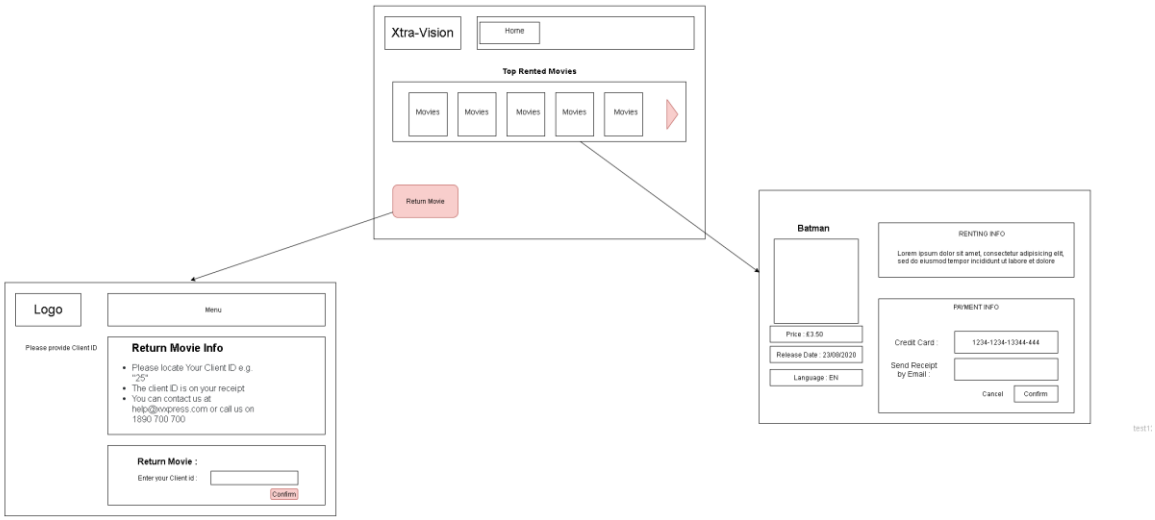
2. CLASS DIAGRAM



3. ACTIVITY DIAGRAM



Wire Frame



CLASSES OF KIOSK SYSTEM

4. CLIENT:

This class is responsible for storing all user information once a movie is allocated.

4.1. ATRIBUTES OF CLIENT CLASS

4.1.1. CLIENT_ID

The client_id is responsible for storing the id that identifies the user who will be renting the movie.

4.1.2. MOVIES_ID

The movies_id attribute stores the movie's id once it has been rented.

4.1.3. MOVIE

The Movie attribute references the Movie Object. Once this object is instantiated, it is possible to access its methods and attributes.

4.1.4. FULL_NAME

The full_name attribute of type string is responsible for storing the full name of the user / customer once a movie is allocated.

4.1.5. CREDIT_CARD

The credit_card attribute is responsible for storing the card number of the user who allocates the movie.

4.1.6. EMAILID

the email id attribute is responsible for storing the customer's email.

4.1.7. RETURNED

The returned attribute stores the status of the movie.

4.2. **FUNCTIONS/METHODS IN CLIENT CLASS**

4.2.1. GETCLIENTID ()

The method when invoked returns the client ID

4.2.2. GETMOVIEID ()

The method when invoked returns the movie id.

4.2.3. GETFULL_NAME ()

The get_fullname method when invoked returns the user's full name.

4.2.4. GETCREDIT_CARD ()

The getCredit_Card method when invoked returns the number of the user's credit card.

4.2.5. GETEMAILID ()

The getEmailId method when invoked returns the user's email.

4.2.6. ISRETURNED ()

The method when invoked returns the status of the movie, being true or false.

5. MOVIES

The movie class is responsible for storing all the information of the rented film such as: Id, title, release date, original language, poster path, and status.

5.1. ATTRIBUTES OF MOVIES CLASS

5.1.1. ID

The id is responsible for storing the movie id

5.1.2. TITLE

The title attribute is responsible for storing the movie title

5.1.3. RELEASE_DATE

The release_date attribute as the name already says, store the release date of the movie.

ORIGINAL_LANGUAGE

The original_language attribute is responsible for storing the original language of the movie rented.

5.1.4. POSTER_PATH

The Poster_path attribute will be responsible for storing the movie's url once the user clicks on the movie's banner.

5.1.5. MOVIE_STATUS

The movie_status attribute stores the status of the movie, whether it is available for rental or not.

5.2. **FUNCTIONS/METHODS IN MOVIE CLASS**

5.2.1. GET_ID ()

The method when invoked returns the movie id.

5.2.2. GETTITLE ()

The method when invoked returns the title of the film.

5.2.3. GETRELEASE_DATE ()

The method when invoked returns the release date of the film.

5.2.4. GETORIGINAL_LANGUAGE ()

The getOriginal_Language method when invoked returns the original language of the movie.

5.2.5. GETPOSTER_PATH ()

The getPoster_Path method when invoked returns the path of the selected movie's banner.

5.2.6. GETMOVIE_STATUS ()

The method getMovie_Status of type boolean when invoked returns whether the movie is available or not (true or false).

6. RETURNMOVIE

The ReturnMovie class will store all the return details, once the customer returns a DVD to the Kiosk System.

6.1. ATTRIBUTES OF RETURNMOVIE CLASS

6.1.1. CLIENTID

The clientId attribute is responsible for storing the client id

6.1.2. MOVIEID

The movieId attribute is responsible for storing the movie's id.

SPRING BOOT

7. CLASSES

7.1. COM.MOVIEKIOSKICRM

A package was created with the name of **com.moviekioskicrm**, within that package it contains the main class that will be the first to be executed and that invokes the other classes and objects.

```java
package com.moviekioskicrm;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class MovieKioskiApplication {


    //private static final Logger log = LoggerFactory.getLogger(MovieKioskiApplication.class);


    public static void main(String[] args) {
        SpringApplication.run(MovieKioskiApplication.class, args);

    }


}
```

7.2. COM.MOVIEKIOSKICRM.CONTROLLER – CLIENTCONTROLLER

A package was created with the name of com.moviekioskicrm.controller, within this package there is a class called ClientController

```java
@CrossOrigin(origins = "http://localhost:3000")
@RestController
@RequestMapping("/movie-api/v1/")
public class ClientController {

    @Autowired
    private RentedMoviesRepository rentedMoviesRepository;


    @PostMapping("/renting") //save movie on database after being rented
    public Clients rentMovies(@RequestBody Clients client) {

        //Movies movie = movieRepository.findById(client.get());

        //send email

        return rentedMoviesRepository.save(client);
    }

    //get by id client and return movie data to front end
    @GetMapping("/client/{id}")
    public Clients getClientById(@PathVariable("id") long id) {

        Optional<Clients> clientsOptional = rentedMoviesRepository.findByIdAndReturnedIsFalse(id);
```

In the Client Controller class, the following annotations were used:

@CrossOrigin: (origins=" http ://localhost:3000")

As the name says is the origin of the web server, without the end points.

@RestController:

Class with the REST endpoint

@RequestMapping:("/movie-api/v1/")

REST endpoint method that assigns an end point to the end of the url

@AutoWired:

This note forces the dependency injection. In other words, it assigns an object to a variable. In the case of the figure above, the RentedMoviesRepository object is being assigned to a variable ***rentedMoviesRepository.***

@PostMapping: ("/renting")

Annotation for submitting data in the database.

@requestBody : Clients client

This annotation maps the body of the Http request to a transfer or domain object. In this case, the object is the client.

@GetMapping: ("/client/{id}")

This annotation takes the end point of the url client

```
public Clients getClientById(@PathVariable("id") long id) {
```

@PathVariable

This annotation allows the specification of the desired field in a url to obtain a certain type of data.

In this class, a method was created to identify whether a particular customer returned the film. If so, the method changes the status in the database to true.

7.3. COM.MOVIEKIOSKICRM.CONTROLLER – MOVIECONTROLLER

In the MovieController class, the same annotations as the previous class were used. However, in this class, a list-type method was created in which all the films in the database are returned.

```
@RequestMapping("/movie-api/v1/")
public class MovieController {

    @Autowired
    private MovieRepository movieRepository;

    private static List<Movies> movieData = new ArrayList<Movies>();

    //get all movies from database and send to front end
    @GetMapping("/movies")
    public List<Movies> getAllMovies(){

        return movieRepository.findAll();

    }
```

7.4. COM.MOVIEKIOSKICRM.MODEL – CLIENTS

In the Clients class, JPA hibernate annotations were used to map the database, they are:

@ Entity

This annotation specifies that the class is an entity and can be stored in the database via ORM.

@Table

The table annotation specifies the details of the table that will be used to persist the entity in the database.

@Id

The @id annotation allows a primary key to be created in the database.

@GeneratedValue(strategy = GenerationType.IDENTITY)

This annotation allows the primary key to be automatically created by the database.

@Column

This annotation creates columns in the database.

Basically this class and the scope for creating the database.

In this class, methods were also created to obtain the user's details, as can be seen in the figure below.

```
    public long getClientId() {
        return client_id;
    }

    public void setClientId(long client_id) {
        this.client_id = client_id;
    }

    public Movies getMovies() {
        return movies;
    }

    public void setMovies(Movies movies) {
        this.movies = movies;
    }

    public String getFull_name() {
        return full_name;
    }

    public void setFull_name(String full_name) {
        this.full_name = full_name;
    }
```

7.5. COM.MOVIEKIOSKICRM.MODEL – MOVIES

In this class, basically the same thing was done for the client class, but with different data.

This class served as the scope for the elaboration of the database, just as it was done in the Client class of the model package.

```
import javax.persistence.Column;

@Entity
@Table(name = "movies")
public class Movies {


    //@GeneratedValue(strategy = GenerationType.IDENTITY)
    @Id
    private long id;

    @Column(name = "title")
    private String title;

    @Column(name = "release_date")
    private String release_date;

    @Column(name = "original_language")
    private String original_language;

    @Column(name = "poster_path")
    private String poster_path;

//  @Column(name = "movie_status")
//  private String movie_status;

    public Movies() {
```

7.6. COM.MOVIEKIOSKICRM.REPOSITORY - MOVIEREPOSITORY

The repository package is where the magic happens. The classes contained within this repository instantiate the classes in the model package summarizing it and responsible for creating the database. Just use the **@Repository** annotation and implement the **JpaRepository** interface as can be seen in the figure below:

```
1  package com.moviekioskicrm.repository;
2
3  import org.springframework.data.jpa.repository.JpaRepository;
4  import org.springframework.stereotype.Repository;
5
6  import com.moviekioskicrm.model.Movies;
7
8  @Repository
9  public interface MovieRepository extends JpaRepository<Movies, Long> {
10
11 }
12
```

7.7. COM.MOVIEKIOSKICRM.REPOSITORY - RENTEDMOVIEREPOSITORY

The same thing was done for the rentedMovieRepository class.

```
1  package com.moviekioskicrm.repository;
2
3  import java.util.Optional;
4
5  import org.springframework.data.jpa.repository.JpaRepository;
6  import org.springframework.data.jpa.repository.Query;
7  import org.springframework.data.repository.query.Param;
8  import org.springframework.stereotype.Repository;
9
10 import com.moviekioskicrm.model.Clients;
11
12
13 @Repository
14 public interface RentedMoviesRepository extends JpaRepository<Clients, Long> {
15
16     @Query(value = "SELECT c FROM Clients c WHERE c.returned = false AND c.client_id = :client_id")
17     Optional<Clients> findByIdAndReturnedIsFalse(@Param("client_id") long clientId);
18
19 }
20
```

8. MAILSENDER

This package was created specifically to house the EmailConfiguration class.

8.1. EMAILCONFIGURATION CLASS

The EmailConfiguration class allows the user to receive confirmation and receipt via email once a film is returned or rented. For this reason, it has annotaion different from the previous classes, they are:

@Value:

This annotation is used to inject values into certain variables. In this case associating with the Application.properties file. See the figures below.

```java
package com.moviekioskicrm.mailsender;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class EmailConfiguration {

    @Value("${spring.mail.username}")
    private String email;

    @Value("${spring.mail.password}")
    private String pass;

    @Value("${spring.mail.host}")
    private String host;

    @Value("${spring.mail.port}")
    private int port;


    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPass() {
        return pass;
```

```
spring.application.name=movie-kiosk
spring.datasource.url=jdbc:mysql://localhost:3306/movies?useSSL=false
spring.datasource.username=root
spring.datasource.password=

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto = update

spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.statttls.required=false
spring.mail.properties.mail.smtp.auth=true

#gmail settings
spring.mail.host=smtp.mailgun.org
spring.mail.port=587
spring.mail.username=postmaster@sandbox50f645a77d804e08b8698bf69ccecb9e.mailgun.org
spring.mail.password=4686c3dec7e604787b0aba424d3db05e-602cc1bf-7700a246

#security gmail settings


spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=5000
spring.mail.properties.mail.smtp.writetimeout=5000
spring.mail.test-connection=true
```

9. APPLICATION.PROPERTIES

Inside the project it contains a file called **application.properties** which is inside **src / main / resources -> META-INF**

This file is where the database settings are located, that is, it is responsible for establishing the connection with the database

.

```
spring.application.name=movie-kiosk
spring.datasource.url=jdbc:mysql://localhost:3306/movies?useSSL=false
spring.datasource.username=root
spring.datasource.password=

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto = update

```

FRONT-END USING REACT

The front-end of the application was made in React.

React is a JavaScript library that was used to create the user interface, mainly for the implementation of the movies API.

10. FOOTER

```
1  import React, { Component } from 'react';
2
3  class FooterComponent extends Component { //this react component render the footer on the browser
4      render() {
5          return (
6              <div>
7                  <footer className="footer">
8                      <span className="text-muted">All Rights Reserved @Jose Paulo || @Henrique Demetrio</span
9                  </footer>
10             </div>
11         );
12     }
13 }
14
15 export default FooterComponent;
```

The figure above shows the source code of the footer. This react component render the footer on the browse.

For that, it was necessary to import React and the Component from React class. The **FooterComponent** and the name of the class that represents the website's footer. After creating the footer class, we inherit the functions and properties of the Component class, using the term extends.

In this class, the **render ()** method was implemented, which returns null or adds HTML. In that case the team's copyright is returning.

Finally, the method class that returns the footer of the page is exported, using the term **export default** followed by the *class name*.

11. MENU BAR

The bar menu is one of the most important components of an application. And where the user will navigate in order to achieve their goal.

```
import React, { Component } from 'react';
import Navbar from "react-bootstrap/Navbar";
import Nav from "react-bootstrap/Nav";
import Button from "react-bootstrap/Button";


class HeaderComponent extends Component { //this react component render the navbar on the broweser
    render() {
        return (
            <div>
                <header>
                    <Navbar bg="dark" variant="dark">
                        <Nav className="mr-auto">
                            <Navbar.Brand href="/">Xtra-Vision</Navbar.Brand>
                            <Nav.Link href="/"><Button variant="primary">Home</Button></Nav.Link>
                        </Nav>
                    </Navbar>
                </header>
            </div>
        );
    }
}

export default HeaderComponent;
```

As shown in the figure above. We started with the import of the **Component, NavBar, Nav, Button** classes so that their attributes could be used.

The principles for the elaboration of the Menu Bar and the same that was done in the **Footer** class, were first identified as **HeaderComponent** and then we inherited the *Component* class to use its functions and properties.

The only difference is that the **render ()** method returns elements other than the Footer class. In the **HeaderComponent** class, the **render ()** method returns the button component, name of the Xtra-Vision system and defines the page background.

Then we export the class using the term export default followed by the class name (HeaderComponent).

12. LIST MOVIES

This class is responsible for manipulating the data of the films and sending them to the database.

The carousel implemented for the films was made with the Swiper, which is responsible for the transition of the films.

```
1  import React, { Component } from 'react';
2  import MoviesService from '../services/MoviesService';
3  import { Swiper, SwiperSlide } from 'swiper/react';
4  import SwiperCore, { Navigation, Pagination, Controller, Thumbs } from 'swiper';
5  import 'swiper/swiper-bundle.css';
6  import Card from "react-bootstrap/Card";
7  import Button from "react-bootstrap/Button";
8  import Badge from "react-bootstrap/Badge";
9
10
11 SwiperCore.use([Navigation, Pagination, Controller, Thumbs]);
```

13. RENT MOVIE

This class is responsible for manipulating the data that will be information by the user. Once the user informs it. The class will save all customer details and the movie that was rented to the database.

```
import React, { Component } from 'react';
import MoviesService from '../services/MoviesService';
import Card from "react-bootstrap/Card";
import Modal from "react-bootstrap/Modal";
import Button from "react-bootstrap/Button";
import {Form, Row, Col, Alert} from "react-bootstrap";




class RentMovie extends Component {
    constructor(props){
        super(props)

        this.state = { //this fields are used to retrieve the movies from the array and sent it to the backe
            movie: [],
            full_name: '',
            credit_card: '',
            emailId: '',
            movies:{
                id: '',
                title: '',
                release_date: '',
                original_language: '',
                poster_path: ''

            },
            errors: {}
        }

        this.getFullNameHandler    = this.getFullNameHandler.bind(this); //bind the method so i can use insid
        this.getCreditCardHandler = this.getCreditCardHandler.bind(this);
```

14. RETURN MOVIE

The Return Movie class handles the rented movie data.

```
import React, { Component } from 'react';
import MoviesService from '../services/MoviesService';
import Button from "react-bootstrap/Button";
import Card from "react-bootstrap/Card";
import Modal from "react-bootstrap/Modal";
import {Form, Row, Col} from "react-bootstrap";

class ReturnMovie extends Component {
    constructor(props){
        super(props)

        this.state = { //this fields get the movie when the clien enter their id on the browser and send it
            client_movie: {},
            isLoading: true,
            error: null,

            full_name: '',
            credit_card: '',
            emailId: '',
            client_id:'',
            movies:{
                id: '',
                title: '',
                release_date: '',
                original_language: '',
                poster_path: ''

            }


        }
```

15. MOVIES SERVICE

The **Movies Service** class is responsible for storing the database settings. The class establishes the connection between the web page and the database.

```
import axios from 'axios';

const MOVIE_API_BASE_URL    = "http://localhost:8080/movie-api/v1/movies";
const MOVIE_API_CLIENT_REQUEST_URL = "http://localhost:8080/movie-api/v1/renting";
const MOVIE_API_CLIENT_RETURN_MOVIE_URL = "http://localhost:8080/movie-api/v1/client/";
const MOVIE_API_CONFIRM_RETURNED_MOVIE = "http://localhost:8080/movie-api/v1/return/movies"

class MoviesService {

    getMovies(){ //this method get all the movies from the Spring Boot Api and return an json object with al
        return axios.get(MOVIE_API_BASE_URL);
    }

    rentedMovie(rentedMovie){ //this method send the client information toghether with the rented movie to t
        return axios.post(MOVIE_API_CLIENT_REQUEST_URL, rentedMovie);
    }

    getClientMovies(client_id){ //this method get the client by id and the movie that has been rented
        return axios.get(MOVIE_API_CLIENT_RETURN_MOVIE_URL + client_id);
    }

    confirmReturnedMovie(client_movie){ //this method confirm when the client return a movie and update the
        return axios.post(MOVIE_API_CONFIRM_RETURNED_MOVIE, client_movie);
    }

}

export default new MoviesService();
```