**WEB APPLICATION FOR STORING AND ADMINISTRATING A MOVIE COLLECTION**

**Authors**: Musteață Andrei, Mircea Adrian

**Coordinating teacher**: Lect. Univ. Dr. Liana Boca, Asist. Univ. Drd. Muntean Maria

**University**: "1 Decembrie 1918" University of Alba Iulia, Faculty of Sciences and Engineering, Computer Science specialization

**ABSTRACT**

*"Front-end" and "back-end" are two common buzzwords used in discussions about app development. They basically divide the task of an app developer into two categories, as a front-end developer and a back-end developer demand different sets of expertise. In order to shed light on this topic of app development, we will have a look at both front-end and back-end development and see how they differ and work together.*

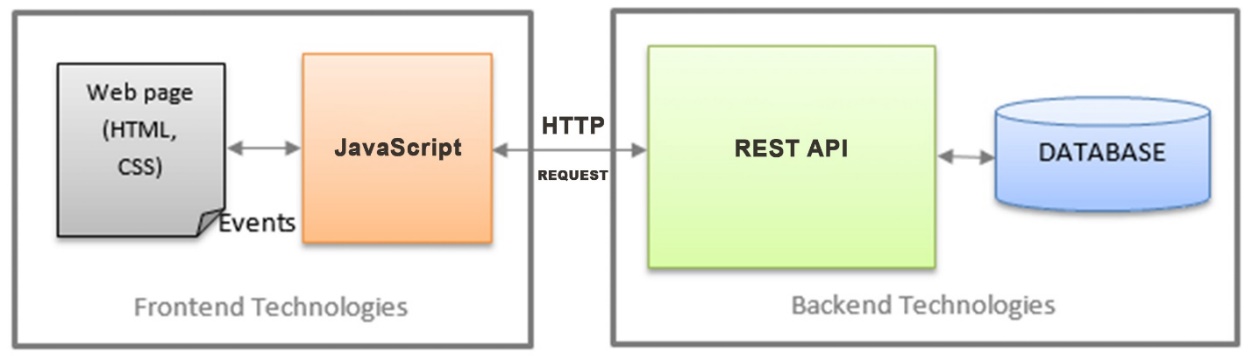
*Front-end and back-end developments are very different, but both are important players in the same game. The front-end would not work effectively without a proper back-end architecture. And without a front-end, the back-end would be impractical. Both are in constant communication with one another to ensure that the user experience is always ideal. This research paper sheds a light on the web programming aspect and other supporting technologies used for the front-end and back-end in web apps.*

# INTRODUCTION

The part of a website that the user interacts with directly is termed the **front end**. It is also referred to as the ‘client side’ of the application. It includes everything that users experience directly: text colors and styles, images, graphs and tables, buttons, colors, and navigation menu. HTML, CSS, and JavaScript are the languages used for Front End development. **Backend** is the server-side of the website. It stores and arranges data, and also makes sure everything on the client-side of the website works fine. It is the part of the website that you cannot see and interact with. It is the portion of software that does not come in direct contact with the users. The parts and characteristics developed by backend designers are indirectly accessed by users through a front-end application. Activities, like writing APIs, creating libraries, and working with system components without user interfaces or even systems of scientific programming, are also included in the backend.

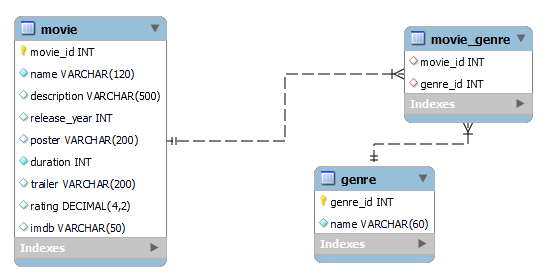
# PROJECT'S PURPOSE

The goal of our project is to develop an online movie library. The client or web interface is accessible through the browser and movies are being stored in a Mysql database via a RESTfull API.



# DATABASE MODEL

First we will build the Mysql Database, here we store all the info related to our application. Each movie includes details such as movie name, description, release year, poster, movie duration, trailer, rating, imdb\_link and movie genre. Because our database is Mysql we can manage and interogate data with SQL (a programming language that enables us to work with that data)



*database diagram*

# BUILDING THE BACKEND REST API

MySQL databases can only store data. If we want to do something with that data, a REST API makes it possible. REST is a method of building and using programs so that they can interact with each other. Without such guidelines, one program could not take information from another and use it.

So our REST API will allow the frontend interface/client to communicate and manage our database.

In building the api we will need to install the following modules :

* **Express.js** – is a web application framework for Node.js. It provides various features that make web application development fast and easy which otherwise takes more time using only Node.js. helps to build efficient and fast web apps.
* **BodyParser** - extracts the entire body portion of an incoming request stream and exposes it on req. This body-parser module parses the JSON, buffer, string and URL encoded data submitted using HTTP POST request.
* **Cors** - Cross-origin resource sharing is a browser security feature that restricts cross-origin HTTP requests that are initiated from scripts running in the browser.

**Main application's code / REST API with Express**

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

const config = require('./config');

const movieRouter = require("./routes/movie");

const cors = require('cors');

app.use(cors({ origin: '\*', optionsSuccessStatus: 200 }));

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.use("/", movieRouter);

app.use("/movies", movieRouter);

app.use("/movie", movieRouter);

app.use((err, req, res, next) => {

  const statusCode = err.statusCode || 500;

  console.error(err.message, err.stack);

  res.status(statusCode).json({ message: err.message });

});

app.listen(config.PORT, () => console.log(`Server is running at port ${config.PORT}`));

**movieRouter** is an express route. An express router refers to how an application’s endpoints (URIs) respond to client request.

**MovieRouter / EXEMPLE OF GETTING ALL THE MOVIES**

const express = require('express');

const router = express.Router();

const db = require('../db');

function emptyOrRows(rows){…}

async function addMovie(movie) {…}

async function updateMovie(movie) {…}

async function deleteMovie({ movie\_id }){…}

async function getMovies() {

  const rows = await db.query(

    `SELECT movie.movie\_id, movie.name, description, release\_year, poster, duration, trailer, rating, imdb, GROUP\_CONCAT(genre.name) AS 'genre' FROM movie

    JOIN movie\_genre on movie.movie\_id = movie\_genre.movie\_id

    JOIN genre on movie\_genre.genre\_id = genre.genre\_id

    GROUP by movie.movie\_id

    ORDER BY movie.movie\_id DESC;`

  );

  const data = emptyOrRows(rows);

  return { data }

}

router.get('/', async function (req, res, next) {

  try {

    const movies = await getMovies(req.query.page);

    res.json({ movies: movies.data });

  } catch (err) {

    console.error(`Error while getting movies `, err.message);

    next(err);

  }

});

router.put('/update', async function (req, res, next) {…}

router.delete('/delete', async function (req, res, next){…}

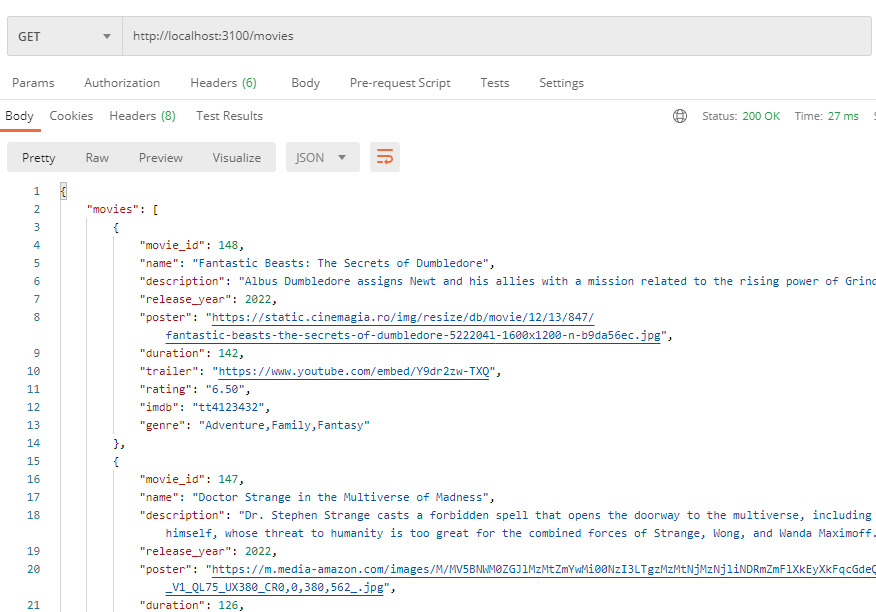
module.exports = router;

We have defined routing using methods of the Express app object that correspond to HTTP methods:

* **router.get()** to handle GET requests – to get all the movies
* **router.post()** to handle POST requests – to add a movie
* **router.delete()** to handle DELETE requests – to delete a movie

When the client makes a GET request on the endpoint "…/**movie(s)**" the server will execute the code wich is found here in MovieRouter. If the endpoint is "**/movie/delete**" the server will execute the function related to "**/delete**" endpoint.

If we look at the example above the expanded function **router.get('/', …** reffers to the endpoint "/movie**/**". Therefore all the code inside will be executed. This function will call getMovies() wich basically executes an SQL querry to get all the movies from the database.res.json **send the resul to the client** in a JSON format.

*testing the api, get request on /movies*

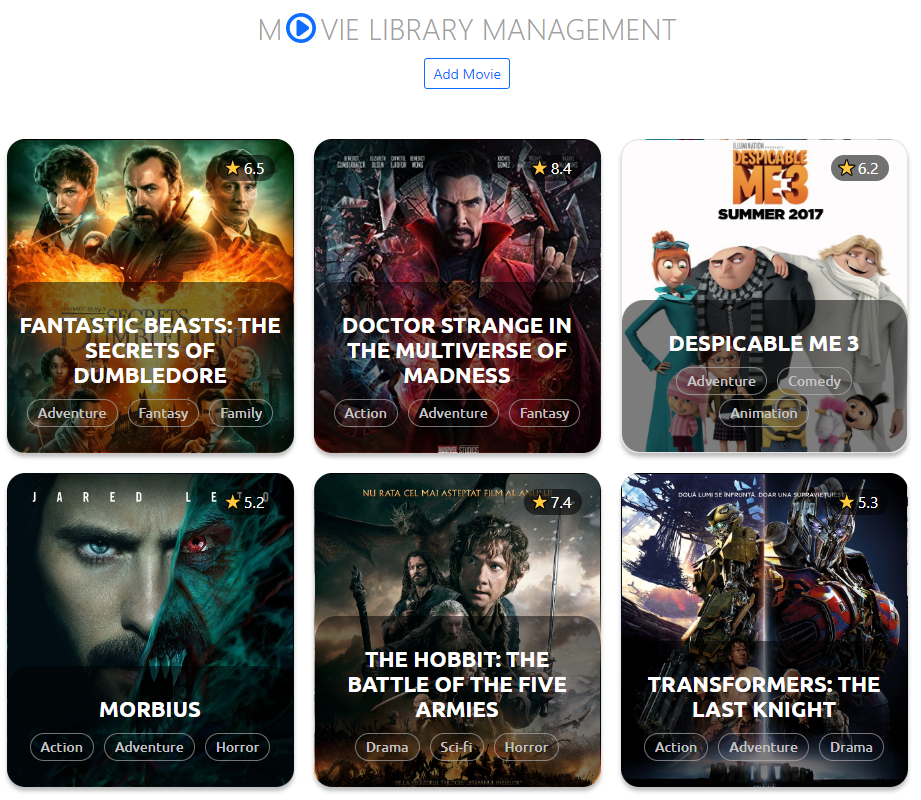
# FRONT END INTERFACE

We have the backend, it works as we expected. But what's the point if there is no

interface to interact with the API.

Therefore we've create a website that contains one single page and 3 popup modals (one for adding a movie, one for editing and one for seeing more details about).

In building the website interface we've used :

* **Html** - language that determines how documents and web pages are displayed in a web browser

*Movie list page*

* **Css** - a popular style sheet language that determines how a document created in HTML is styled (colors, font styles, layout and responsive features).
* **Javascript** – allows you to change CSS and HTML elements on your website after the site has been loaded, which gives you the ability to make your site more interactive and engaging for users.

**app-container HTML**

<div class="app-container">

    <header>

      <div class="logo">

        M<img src="logo.svg" alt="logo" class="logo" />vie Library Management

      </div>

      <button class="btn btn-sm  btn-outline-primary addAndUpdateMovie-btn"

onclick="formReset()" data-bs-toggle="modal"

        data-bs-target="#addAndUpdateMovieModal"

>AddMovie</button>

    </header>

    <div class="movie-list"></div>

  </div>

**CSS of movie card**

.movie {

  width: 287px; height: 314px;

  border: 1px solid rgba(0, 0, 0, 0.11);

  box-sizing: border-box;

  box-shadow: 0px 4px 4px rgba(0, 0, 0, 0.25);

  border-radius: 18px;

  background-size: cover;

  margin: 0 20px 20px 0;

  display: flex; align-items: flex-end;

  position: relative;user-select: none;

  transition-duration: 1s;

}

.movie:hover {

  transform: scale(1.2);

  border: 1px solid #fff;

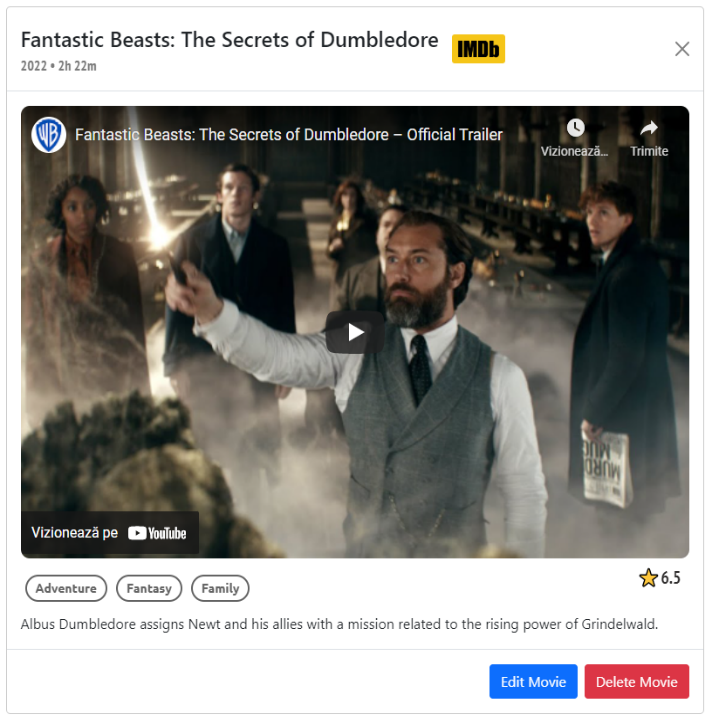
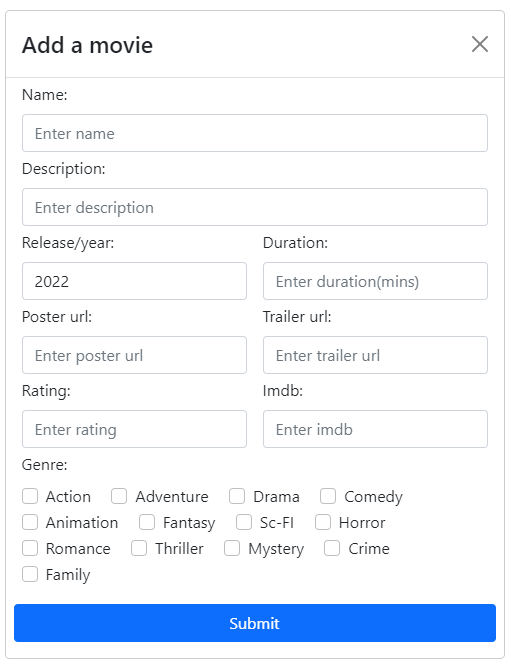
  box-shadow: 0 14px 14px rgba(0, 0, 0, 0.7);

  transition-duration: 0.5s;

  z-index: 999;

}

**Add Movie Modal** – is that part where the client is allowed to add a new movie to the library. The information from dialog's form is being send to the backend api and then procesed to be saved into database(as described in section 4) **Edit Movie Modal** acts similar, but it allows us to edit a movie already created.

****Movie Info Modal** – By selecting a movie a popul will show up with more details, including release year, movie trailer and description. For more info you can go straight to imdb offical site by clicking on that specific logo next to the name

# CLIENT SIDE - HTPP REQUESTS

In orader to pass the information from frontend modals to backend api we have to use HTTP request. This is how we will cominucate with.

*An HTTP request is made by a client, to a named host, which is located on a server. The aim of the request is to access a resource on the server. To make the request, the client uses components of a URL, which includes the information needed to access the resource.*

Our frontend contains 4 of these HTTP requests:

* **GET REQUEST** for getting all the movies
* **POST REQUEST** for adding a movie
* **PUT REQUEST** for editing a movie
* **DELETE REQUEST** for deleting a movie

For example, to get all the movies in the main page the client will:

1. **REQUEST** the API for movies on endpoint (**/movies)**
2. When the server responses back the **frontend'**s DOM elements(HTML) will be **updated** dynamicaly by javascript.

The same process for deleting, except we need to pass to the api the movie id we want to delete. If the the api returns back a coresponding message that means the movie has been sucessfuly removed from database. If that's the case then the coresponding DOM element will be removed from the frontend view.

**Deleting a MOVIE**

const deleteMovie = async movie\_id => {

  const response = await fetch(`${API\_BASE}/movie/delete`, {

    method: 'DELETE',

    headers: {'Content-Type': 'application/json'},

    body: JSON.stringify({ movie\_id }),

  });

  const movie = await response.json();

  if (movie === "DELETED") {

    document.getElementById(`movie\_${movie\_id}`).remove();

  }

}

**Geting the MOVIES**

const getMovies = async () => {

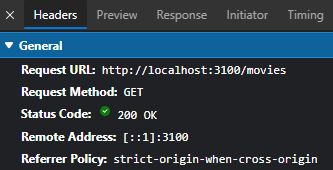
  const response = await fetch(`${API\_BASE}/movies`);

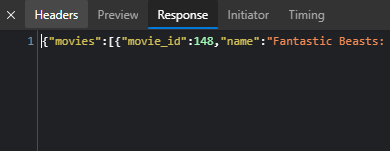
  const { movies } = await response.json();

  document.getElementById('movie-list').innerHTML = movies.map(movie => Movie(movie)).join('');

}

*Request on /movies*



*Server response*

# CONCLUSION

We did it! Our aplication is running live. There was a lot of work on both sides frontend and backend. They are very different tehnologies and both of them needs to function propertly otherwirse the application is down. This was just an example of how these 2 "players" connects togheter in order to serve application's objective. In production there is one team that works with frontend and anothear team with backend. Both teams having the same goal to connect their work together and deliver a complete functional website.

# BIBLIOGRAPHY

[Frontend vs Backend - GeeksforGeeks](https://www.geeksforgeeks.org/frontend-vs-backend/)

[Express.js (tutorialsteacher.com)](https://www.tutorialsteacher.com/nodejs/expressjs)

[HTTP requests - IBM Documentation](https://www.ibm.com/docs/en/cics-ts/5.3?topic=protocol-http-requests)