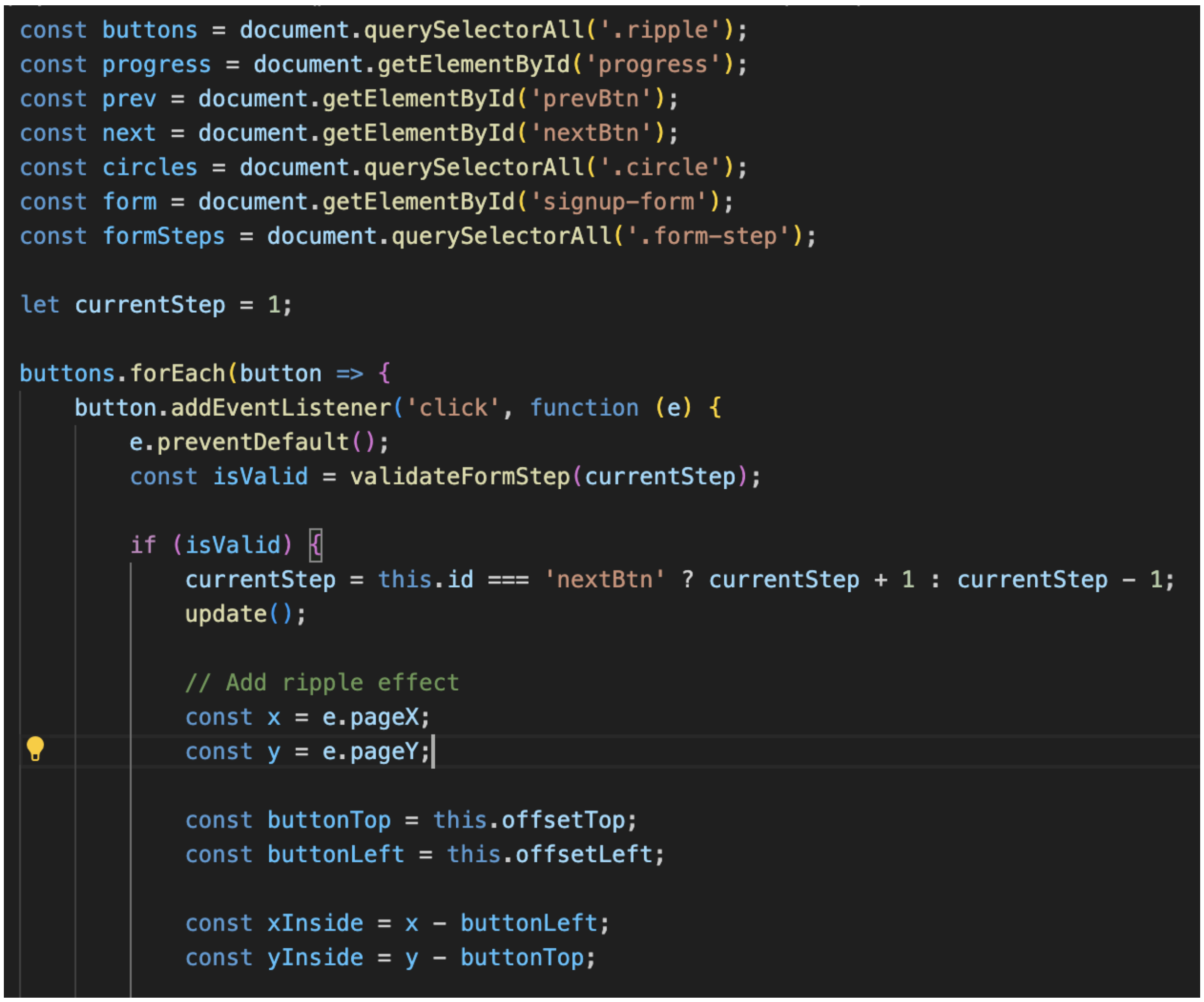
Gray Paper

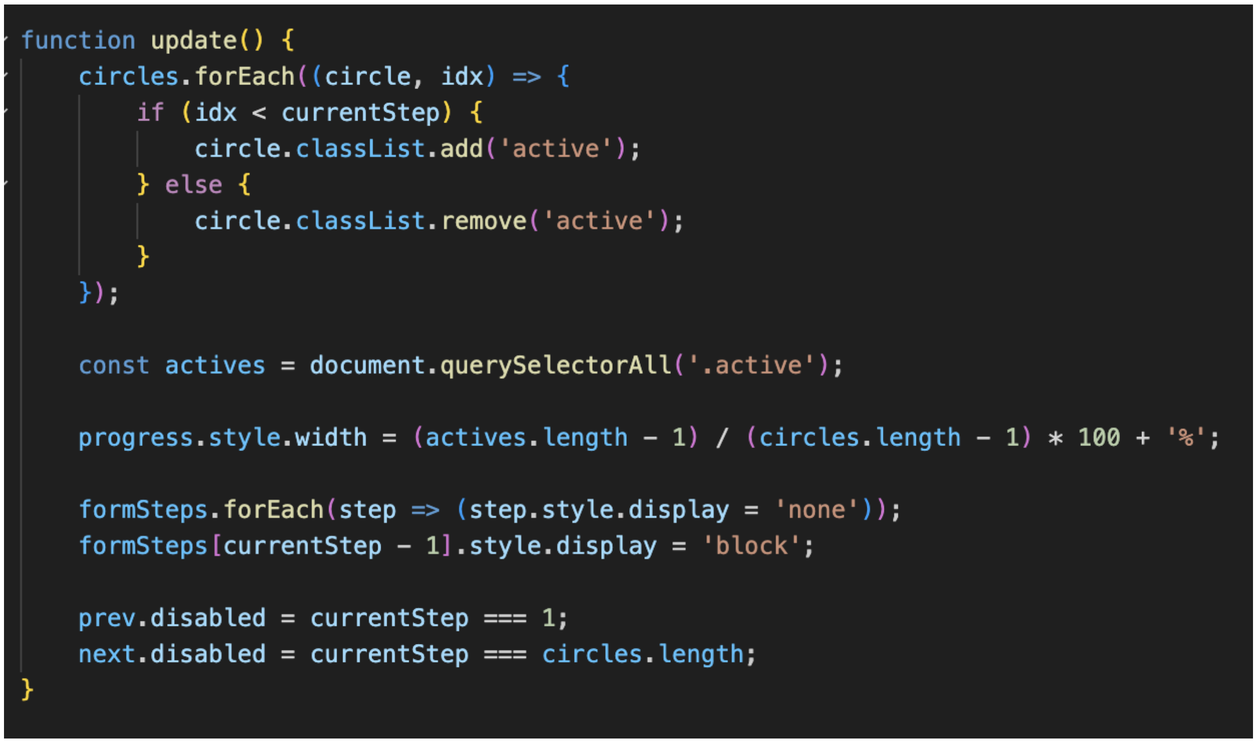
Group Members: Vishal Charran and Ankit Kumar

- ❖ Include at least 10 salient points from the code (including what challenges you encountered and how you solved the problems).

1) The following functions include a progress step and ripple effect for the buttons used inside the sign up page. The progress bar increases each time a user enters an input for the sign up details and when the user hits “Next” or “Prev” a ripple effect can be seen.

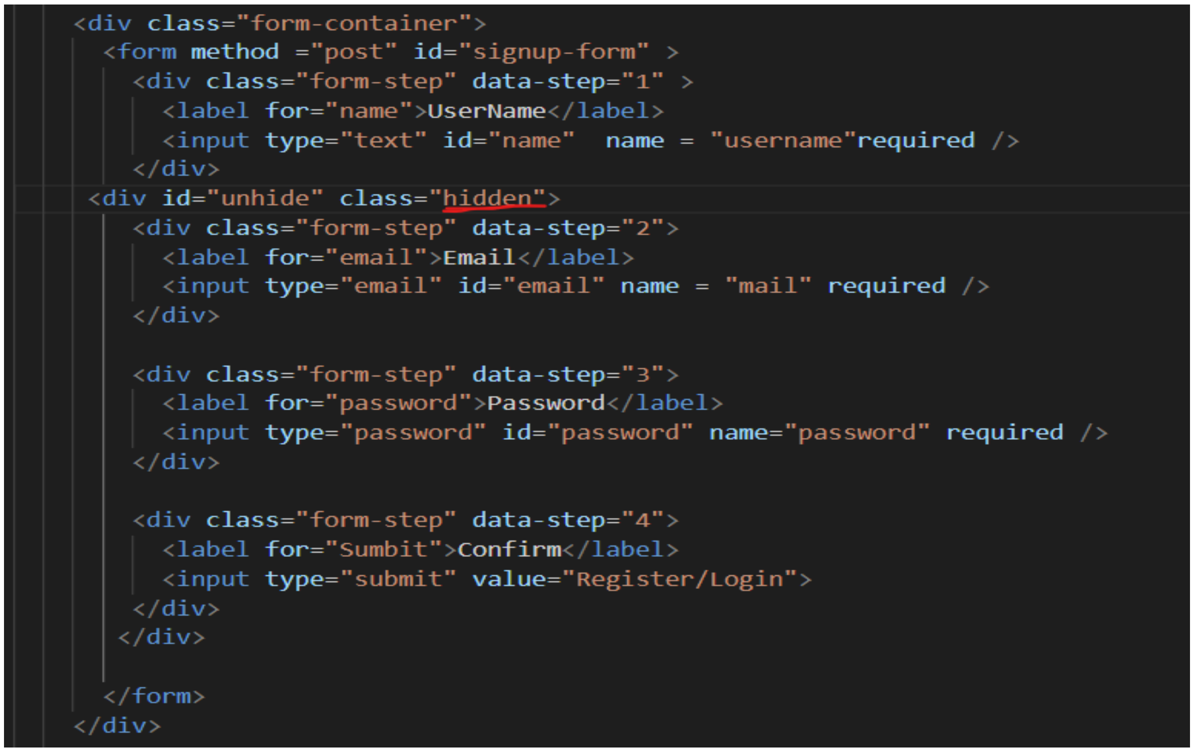


```
const buttons = document.querySelectorAll('.ripple');
const progress = document.getElementById('progress');
const prev = document.getElementById('prevBtn');
const next = document.getElementById('nextBtn');
const circles = document.querySelectorAll('.circle');
const form = document.getElementById('signup-form');
const formSteps = document.querySelectorAll('.form-step');

let currentStep = 1;

buttons.forEach(button => {
    button.addEventListener('click', function (e) {
        e.preventDefault();
        const isValid = validateFormStep(currentStep);

        if (isValid) {
            currentStep = this.id === 'nextBtn' ? currentStep + 1 : currentStep - 1;
            update();

            // Add ripple effect
            const x = e.pageX;
            const y = e.pageY;

            const buttonTop = this.offsetTop;
            const buttonLeft = this.offsetLeft;

            const xInside = x - buttonLeft;
            const yInside = y - buttonTop;
```

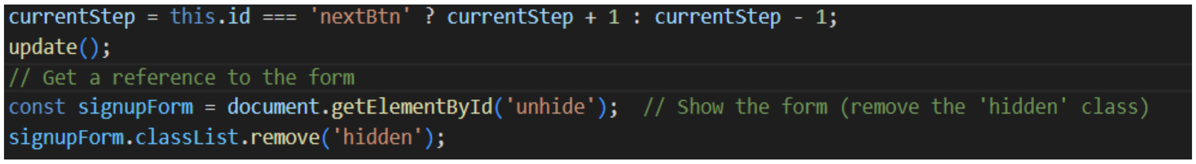


```
function update() {
    circles.forEach((circle, idx) => {
        if (idx < currentStep) {
            circle.classList.add('active');
        } else {
            circle.classList.remove('active');
        }
    });

    const actives = document.querySelectorAll('.active');

    progress.style.width = (actives.length - 1) / (circles.length - 1) * 100 + '%';

    formSteps.forEach(step => (step.style.display = 'none'));
    formSteps[currentStep - 1].style.display = 'block';

    prev.disabled = currentStep === 1;
    next.disabled = currentStep === circles.length;
}
```

2) There was an issue with the entire form as soon as the page loads. Our goal is to have one input box be seen at a time. To fix this issue I hide all input boxes except the first one and unhide them once the user clicks next.

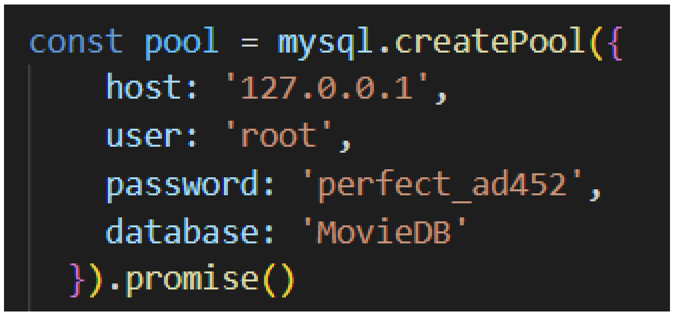


```
    <div class="form-container">
      <form method ="post" id="signup-form" >
        <div class="form-step" data-step="1" >
          <label for="name">UserName</label>
          <input type="text" id="name"  name = "username"required />
        </div>
     <div id="unhide" class="hidden">
        <div class="form-step" data-step="2">
          <label for="email">Email</label>
          <input type="email" id="email" name = "mail" required />
        </div>

        <div class="form-step" data-step="3">
          <label for="password">Password</label>
          <input type="password" id="password" name="password" required />
        </div>

        <div class="form-step" data-step="4">
          <label for="Sumbit">Confirm</label>
          <input type="submit" value="Register/Login">
        </div>
      </div>

      </form>
    </div>
```



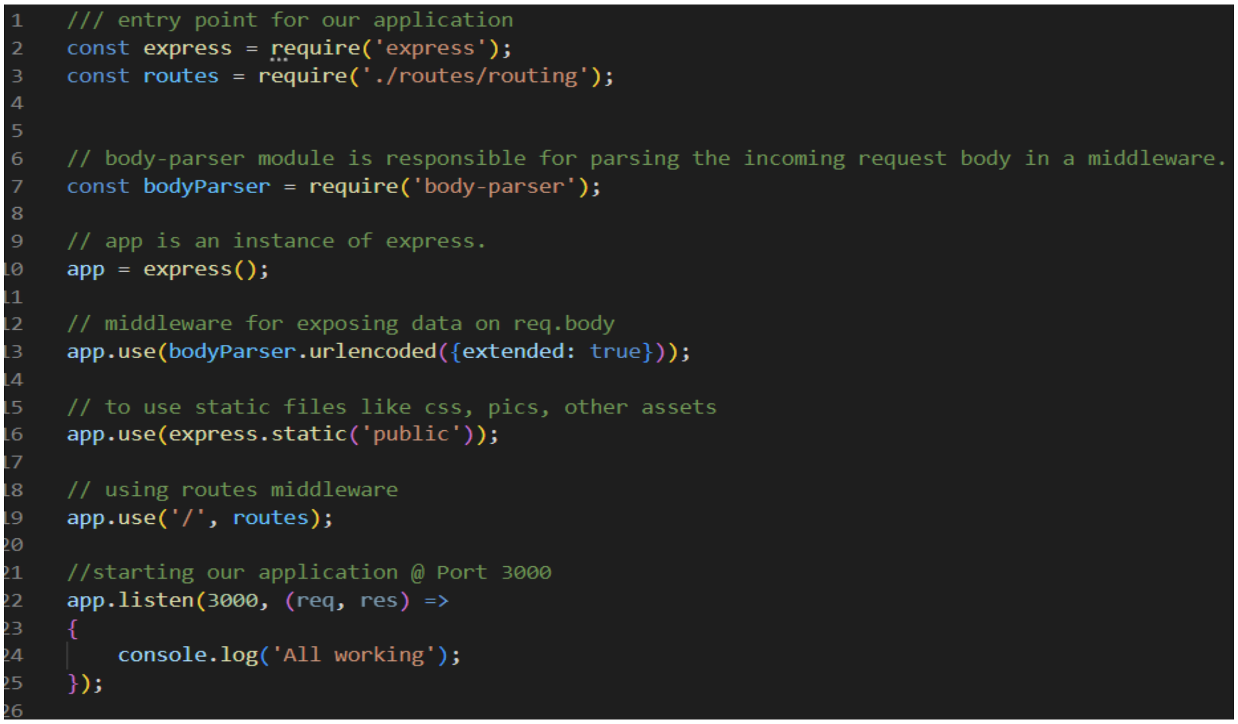
```
currentStep = this.id === 'nextBtn' ? currentStep + 1 : currentStep - 1;
update();
// Get a reference to the form
const signupForm = document.getElementById('unhide');  // Show the form (remove the 'hidden' class)
signupForm.classList.remove('hidden');
```

3) .promise() is used to convert the MySQL connection from a callback-based API to a promise-based API. When using promises, I can use async/await syntax, which can make the code more readable and maintainable compared to using callbacks.



```
const pool = mysql.createPool({
    host: '127.0.0.1',
    user: 'root',
    password: 'perfect_ad452',
    database: 'MovieDB'
  }).promise()
```

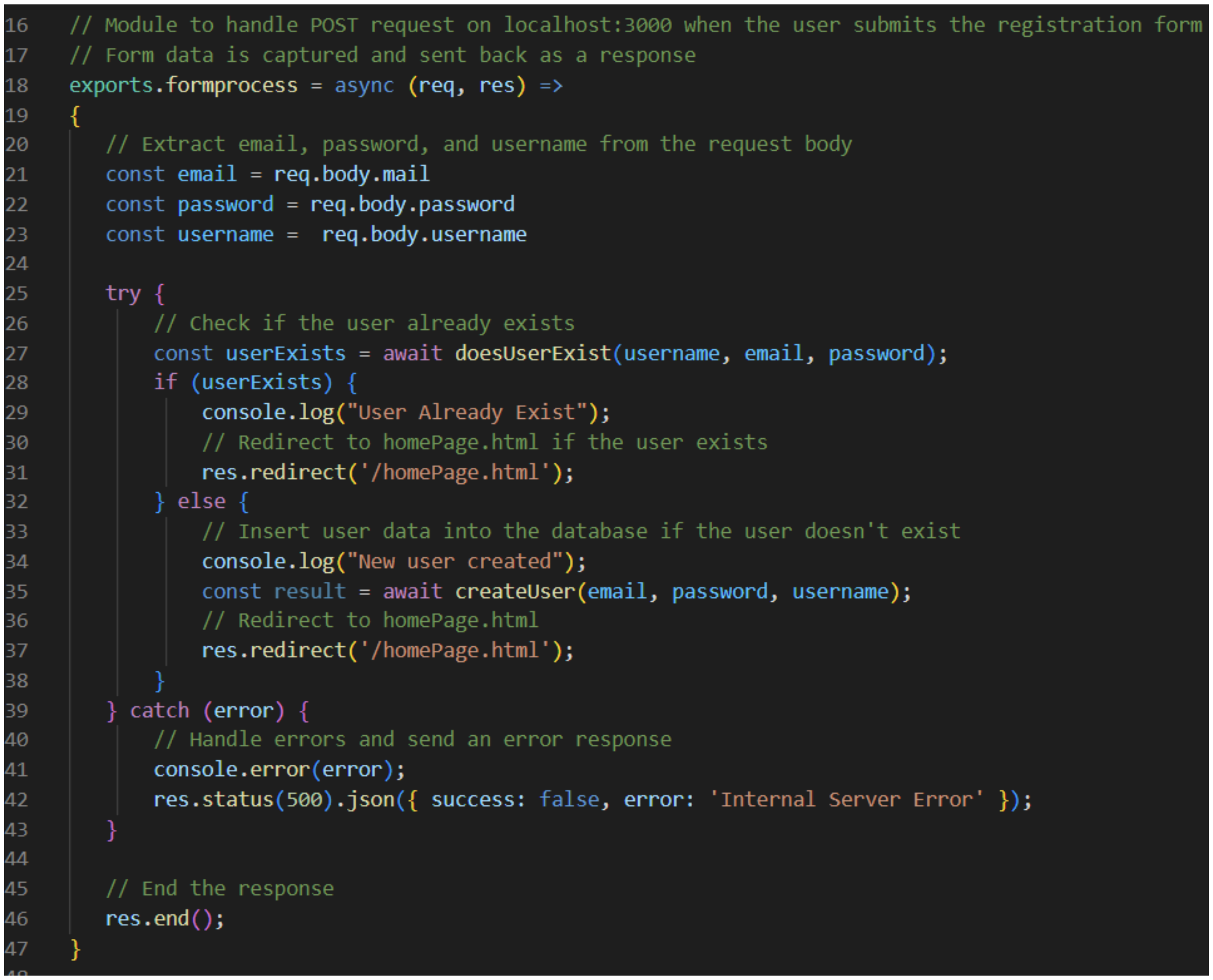
4) This code sets up an Express application, configures middleware for parsing request bodies and serving static files, imports custom routes, and starts the server on port 3000. The routing logic is separated into the ./routes/routing module, which is used to define specific routes and their corresponding handlers.



```
/// entry point for our application
const express = require('express');
const routes = require('./routes/routing');


// body-parser module is responsible for parsing the incoming request body in a middleware.
const bodyParser = require('body-parser');

// app is an instance of express.
app = express();

// middleware for exposing data on req.body
app.use(bodyParser.urlencoded({extended: true}));

// to use static files like css, pics, other assets
app.use(express.static('public'));

// using routes middleware
app.use('/', routes);

//starting our application @ Port 3000
app.listen(3000, (req, res) =>
{
    console.log('All working');
});

```

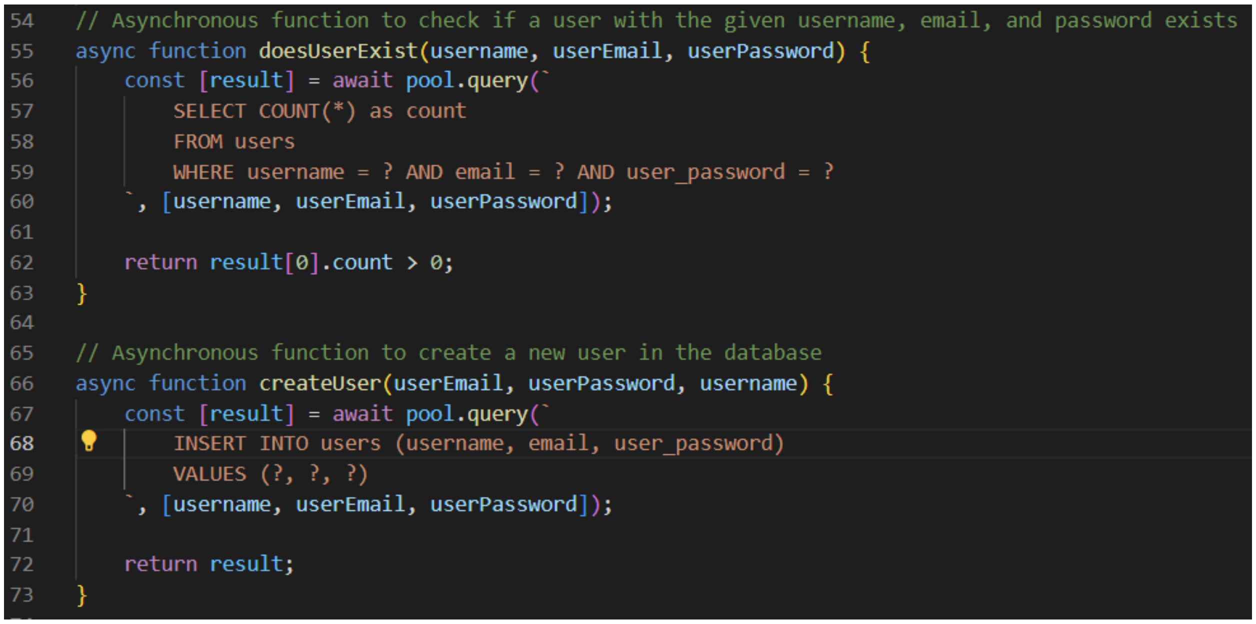
5) The code is a handler for a POST request to localhost:3000 that processes user registration form submissions.

Extracts email, password, and username from the request body.
Checks if a user with the provided username, email, and password already exists using an asynchronous function doesUserExist.
If the user exists, redirects the user to homePage.html.
If the user doesn't exist, inserts the user into the database using an asynchronous function createUser, and redirects the user to homePage.html.
Handles errors by logging them and sending a JSON response with a 500 Internal Server Error status.
Ends the response.
The code ensures proper handling of user registration, checks for existing users, performs database operations asynchronously, and handles errors appropriately. The end result is a redirection to homePage.html after processing the registration form.



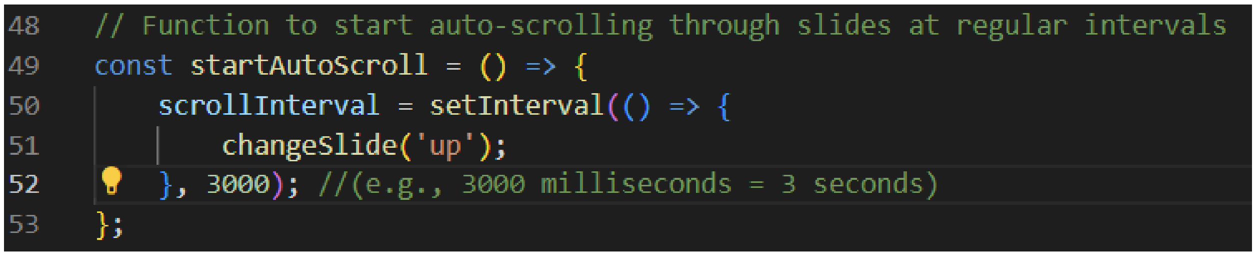
```javascript
// Module to handle POST request on localhost:3000 when the user submits the registration form
// Form data is captured and sent back as a response
exports.formprocess = async (req, res) =>
{
   // Extract email, password, and username from the request body
   const email = req.body.mail
   const password = req.body.password
   const username =  req.body.username

   try {
       // Check if the user already exists
       const userExists = await doesUserExist(username, email, password);
       if (userExists) {
           console.log("User Already Exist");
           // Redirect to homePage.html if the user exists
           res.redirect('/homePage.html');
       } else {
           // Insert user data into the database if the user doesn't exist
           console.log("New user created");
           const result = await createUser(email, password, username);
           // Redirect to homePage.html
           res.redirect('/homePage.html');
       }
   } catch (error) {
       // Handle errors and send an error response
       console.error(error);
       res.status(500).json({ success: false, error: 'Internal Server Error' });
   }

   // End the response
   res.end();
}
```

6) These asynchronous functions are designed to interact with a MySQL database. The doesUserExist function checks for the existence of a user in the database by querying the number of records that match the provided username, email, and password. It returns true if the count is greater than 0, indicating that the user already exists, and false otherwise. On the other hand, the createUser function inserts a new user into the database with the given email, password, and username. It utilizes an asynchronous query to insert the user data into the users table. Both functions leverage the await keyword to handle the asynchronous nature of the database queries, and they assume the existence of a MySQL connection pool (pool). The specific structure of the result object returned depends on the MySQL library being used.



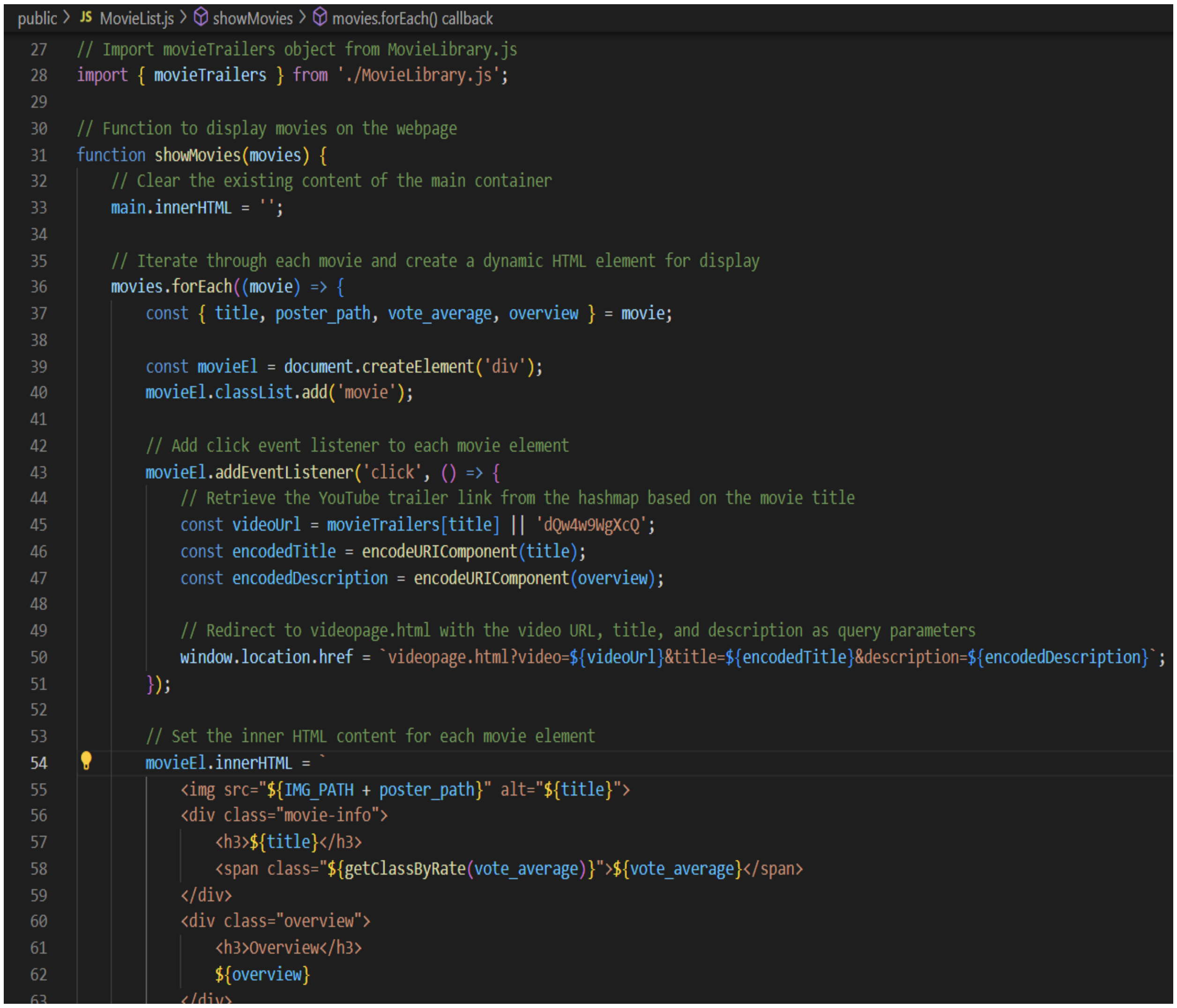
```
// Asynchronous function to check if a user with the given username, email, and password exists
async function doesUserExist(username, userEmail, userPassword) {
    const [result] = await pool.query(`
        SELECT COUNT(*) as count
        FROM users
        WHERE username = ? AND email = ? AND user_password = ?
    `, [username, userEmail, userPassword]);

    return result[0].count > 0;
}

// Asynchronous function to create a new user in the database
async function createUser(userEmail, userPassword, username) {
    const [result] = await pool.query(`
        INSERT INTO users (username, email, user_password)
        VALUES (?, ?, ?)
    `, [username, userEmail, userPassword]);

    return result;
}
```

7) This function, startAutoScroll, initiates an automatic scrolling mechanism through slides at regular intervals. It uses the setInterval function to repeatedly call the changeSlide function with the argument 'up' every 3000 milliseconds (3 seconds). This interval defines the pace at which the slides will automatically transition, providing a continuous and automated scrolling experience for the slider.

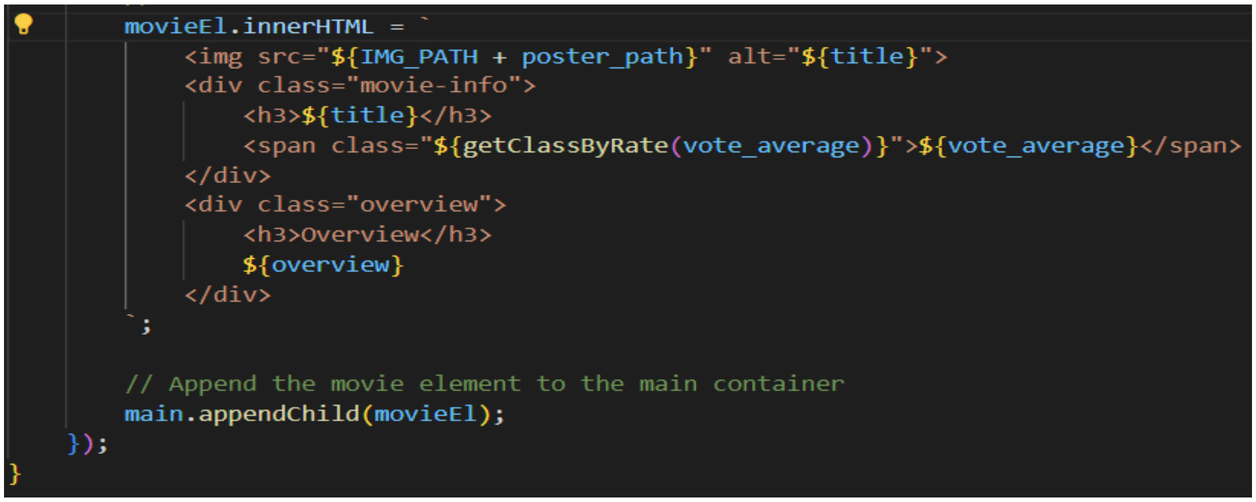


```
// Function to start auto-scrolling through slides at regular intervals
const startAutoScroll = () => {
    scrollInterval = setInterval(() => {
        changeSlide('up');
    }, 3000); //(e.g., 3000 milliseconds = 3 seconds)
};
```

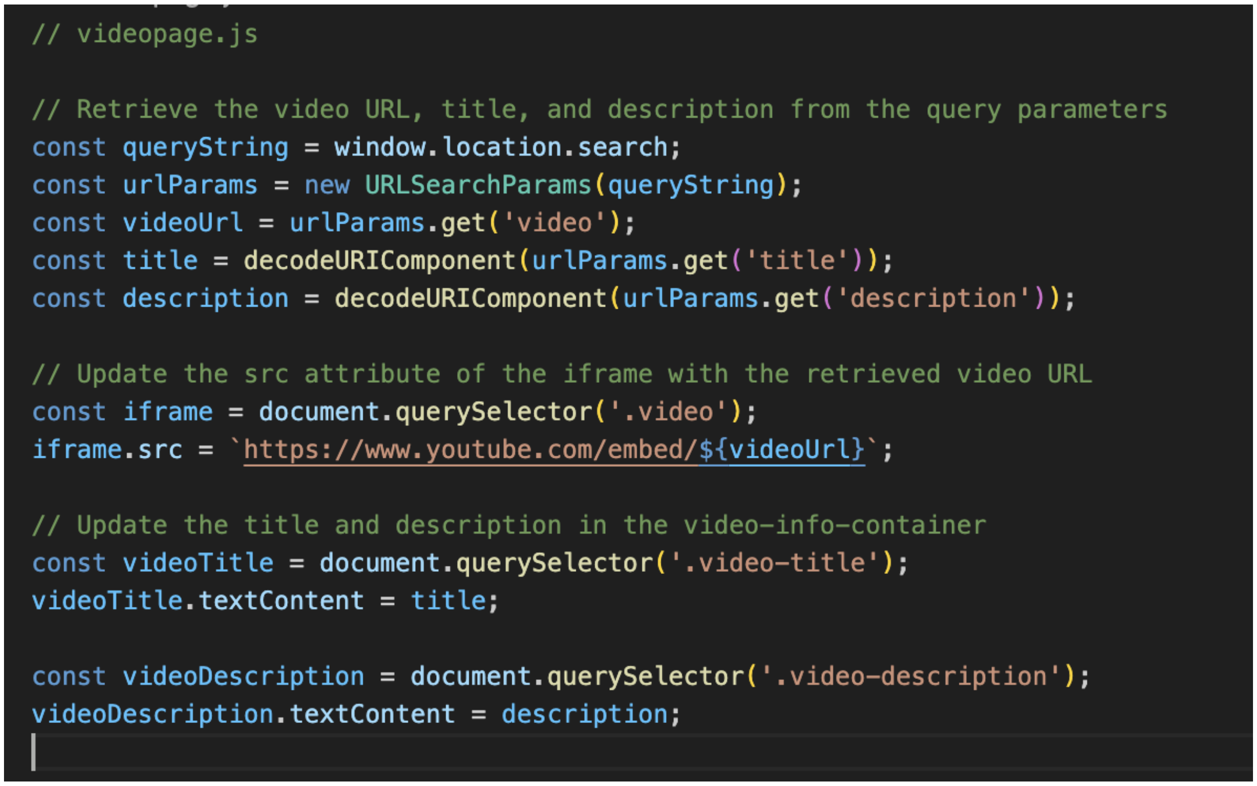
8)The provided JavaScript code defines a function named showMovies designed to showcase an array of movies on a webpage. Importing a movieTrailers object from './MovieLibrary.js', it incorporates YouTube trailer links for various movies. The function begins by clearing the existing content of the HTML element with the id 'main,' presumed to be a container for displaying movies. For each movie in the array, it extracts essential details such as title, poster path, vote average, and overview. The function then creates a dynamic HTML element for each movie, complete with an image, movie information, and an overview. Additionally, a click event listener is attached to each movie element, triggering a redirection to a 'videopage.html' with relevant query parameters upon a click. These parameters include the YouTube trailer link, encoded title, and encoded overview. Overall, the showMovies function efficiently generates and displays movie elements on the webpage, providing an interactive experience for users to explore movie details and trailers.

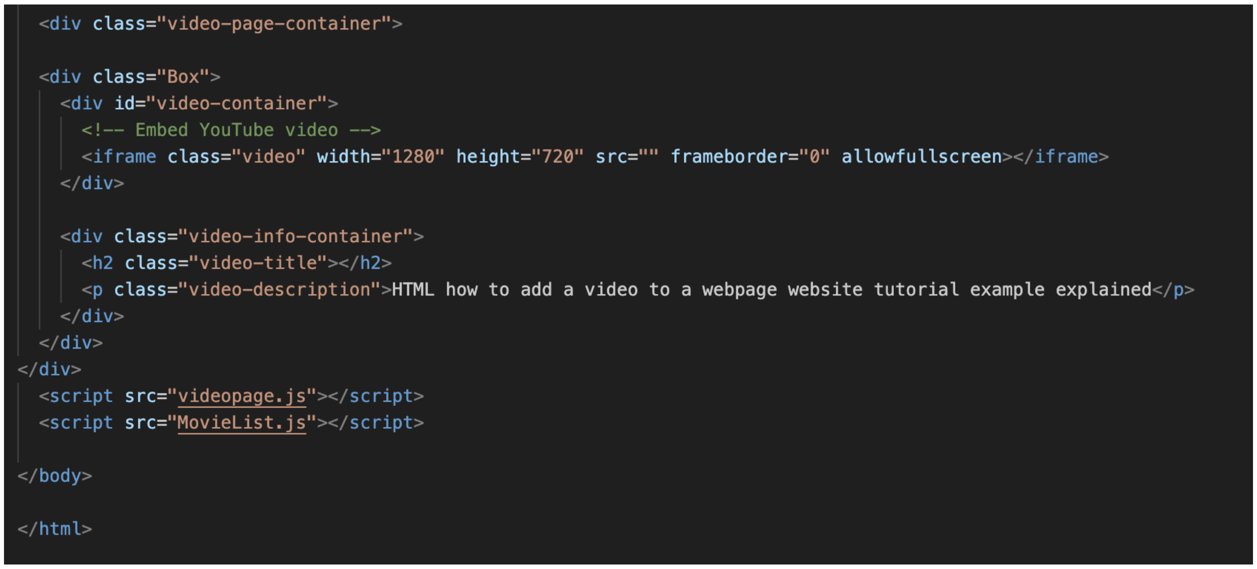

public > JS MovieList.js > showMovies > movies.forEach() callback

```javascript
// Import movieTrailers object from MovieLibrary.js
import { movieTrailers } from './MovieLibrary.js';

// Function to display movies on the webpage
function showMovies(movies) {
    // Clear the existing content of the main container
    main.innerHTML = '';

    // Iterate through each movie and create a dynamic HTML element for display
    movies.forEach((movie) => {
        const { title, poster_path, vote_average, overview } = movie;

        const movieEl = document.createElement('div');
        movieEl.classList.add('movie');

        // Add click event listener to each movie element
        movieEl.addEventListener('click', () => {
            // Retrieve the YouTube trailer link from the hashmap based on the movie title
            const videoUrl = movieTrailers[title] || 'dQw4w9WgXcQ';
            const encodedTitle = encodeURIComponent(title);
            const encodedDescription = encodeURIComponent(overview);

            // Redirect to videopage.html with the video URL, title, and description as query parameters
            window.location.href = `videopage.html?video=${videoUrl}&title=${encodedTitle}&description=${encodedDescription}`;
        });

        // Set the inner HTML content for each movie element
        movieEl.innerHTML = `
            <img src="${IMG_PATH + poster_path}" alt="${title}">
            <div class="movie-info">
                <h3>${title}</h3>
                <span class="${getClassByRate(vote_average)}">${vote_average}</span>
            </div>
            <div class="overview">
                <h3>Overview</h3>
                ${overview}
```


```
        movieEl.innerHTML = `
            <img src="${IMG_PATH + poster_path}" alt="${title}">
            <div class="movie-info">
                <h3>${title}</h3>
                <span class="${getClassByRate(vote_average)}">${vote_average}</span>
            </div>
            <div class="overview">
                <h3>Overview</h3>
                ${overview}
            </div>
        `;

        // Append the movie element to the main container
        main.appendChild(movieEl);
    });
}
```

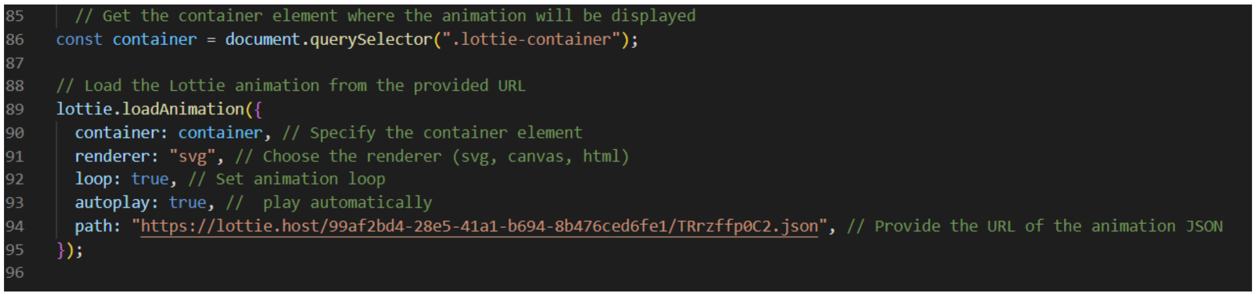
9) This piece of code retrieves the URL, title, and description of the movie the user has selected. The "document.querySelector" automatically retrieves the information and updates it accordingly to display the output. The video area and title/description area is separated by two different forms because at first it was overlapping with the video section which was causing an error and messing with the playback of the movie/video therefore we had to separate them in two different containers to overcome this issue.


```
// videopage.js

// Retrieve the video URL, title, and description from the query parameters
const queryString = window.location.search;
const urlParams = new URLSearchParams(queryString);
const videoUrl = urlParams.get('video');
const title = decodeURIComponent(urlParams.get('title'));
const description = decodeURIComponent(urlParams.get('description'));

// Update the src attribute of the iframe with the retrieved video URL
const iframe = document.querySelector('.video');
iframe.src = `https://www.youtube.com/embed/${videoUrl}`;

// Update the title and description in the video-info-container
const videoTitle = document.querySelector('.video-title');
videoTitle.textContent = title;

const videoDescription = document.querySelector('.video-description');
videoDescription.textContent = description;
```



```html
  <div class="video-page-container">

  <div class="Box">
    <div id="video-container">
      <!-- Embed YouTube video -->
      <iframe class="video" width="1280" height="720" src="" frameborder="0" allowfullscreen></iframe>
    </div>

    <div class="video-info-container">
      <h2 class="video-title"></h2>
      <p class="video-description">HTML how to add a video to a webpage website tutorial example explained</p>
    </div>
  </div>
</div>
  <script src="videopage.js"></script>
  <script src="MovieList.js"></script>

</body>

</html>
```

10) This code uses the Lottie library to display an animation on a webpage. It selects an HTML element with the class "lottie-container" as the container for the animation. The lottie.loadAnimation function is then invoked with a configuration object specifying the container, renderer (in this case, "svg"), animation loop, autoplay, and the URL of the JSON file containing the animation data. The animation will be automatically played in a loop within the specified container on the webpage.



```javascript
  // Get the container element where the animation will be displayed
const container = document.querySelector(".lottie-container");

// Load the Lottie animation from the provided URL
lottie.loadAnimation({
  container: container, // Specify the container element
  renderer: "svg", // Choose the renderer (svg, canvas, html)
  loop: true, // Set animation loop
  autoplay: true, //  play automatically
  path: "https://lottie.host/99af2bd4-28e5-41a1-b694-8b476ced6fe1/TRrzffp0C2.json", // Provide the URL of the animation JSON
});

```

Project By: Ankit Kumar and Vishal Charran