COMP3385 Assignment 4

Due Date:

April 8, 2024 at 11:59 PM

In this assignment, you will practice integrating a VueJS frontend with a Laravel backend API. You will be required to create a basic application that allows you to add your favorite movies to a database and display them on a page.

Learn VueJS

Here are some helpful links to learn about VueJS.

VueJS 3 Guide: https://vuejs.org/guide/

Note

Remember we will be using the Composition API, so ensure in the VueJS documentation that you select that option.

Debugging VueJS

It is recommended that you also install the VueJS Devtools browser extension to help with debugging your VueJS application. You may follow the instructions and download the extension at the following link:

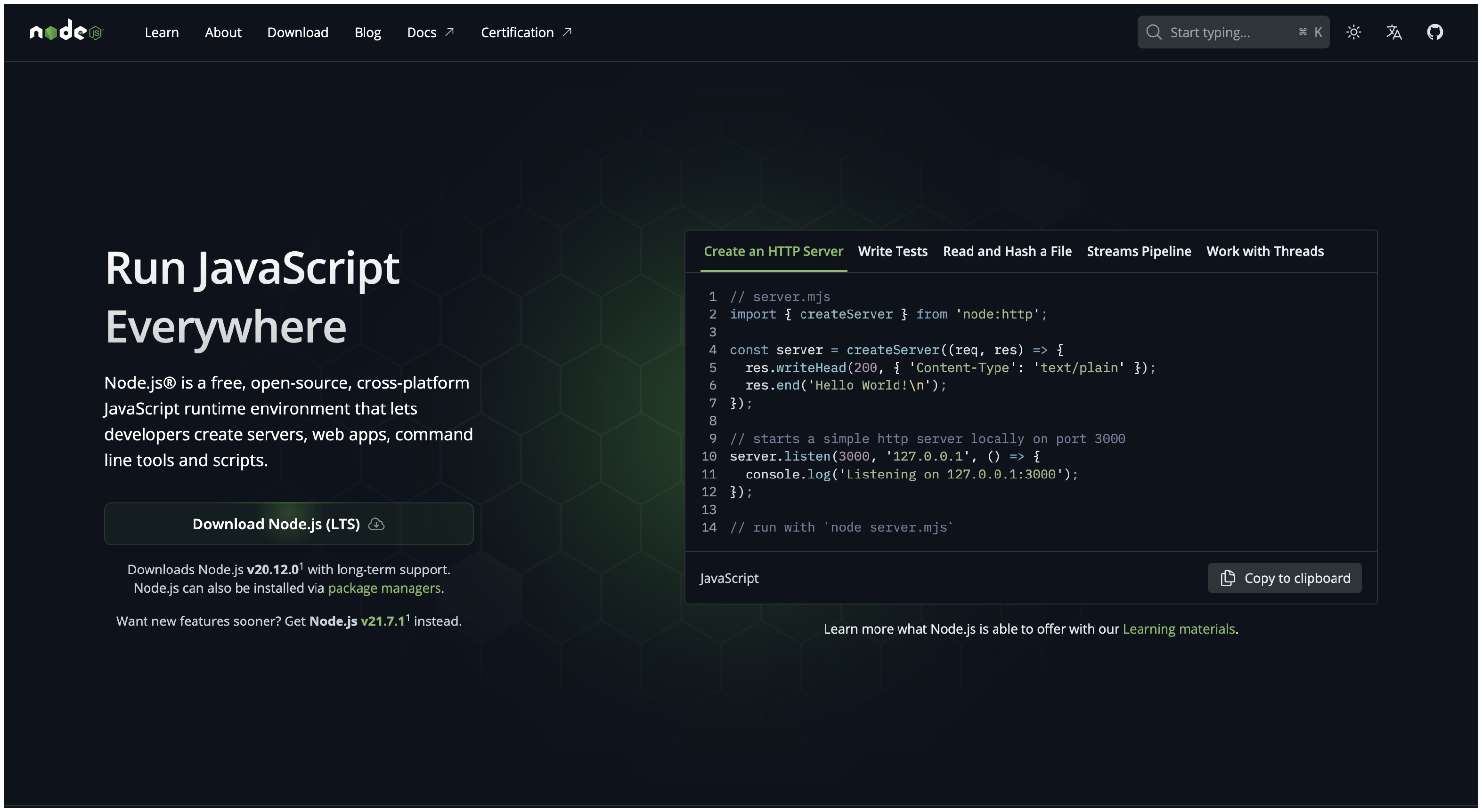
https://devtools.vuejs.org/

Ensure you choose the option for either Google Chrome or Mozilla Firefox.

Exercise 1: Initial Project Setup

Installing NodeJS

Ensure that you download and install NodeJS (if you haven't previously done so) from https://nodejs.org. Choose the option to "Download Node.js (LTS)".



Run JavaScript Everywhere

Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.

Downloads Node.js **v20.12.0**[1] with long-term support. Node.js can also be installed via package managers.

Want new features sooner? Get **Node.js v21.7.1**[1] instead.

```
// server.mjs
import { createServer } from 'node:http';

const server = createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World!\n');
});

// starts a simple http server locally on port 3000
server.listen(3000, '127.0.0.1', () => {
  console.log('Listening on 127.0.0.1:3000');
});

// run with `node server.mjs`
```

Learn more what Node.js is able to offer with our Learning materials.

Once NodeJS is installed, you will need to fork and clone the starter code repository and install the dependencies for your VueJS project and start the development server.

Fork and Clone the Repository

1. Fork the Assignment 4 repository at https://github.com/uwi-comp3385/comp3385-assignment-4.git
2. Then at the command line clone your repository using: `git clone https://github.com/{yourusername}/comp3385-assignment-4.git`
3. Change directories into your new comp3385-assignment-4 folder using `cd comp3385-assignment-4`.
4. Run `composer install` and `npm install` to install the PHP and JavaScript dependencies for this application.
5. Copy `.env.example` and rename it to `.env`.

6. Generate a new `APP_KEY` in your `.env` file by running the artisan command `php artisan key:generate`.

7. Open your `.env` file and edit your `DB_*` settings. For example:

```ini
DB_CONNECTION=pgsql
DB_HOST=127.0.0.1
DB_PORT=5432
DB_DATABASE=comp3385_lab4
DB_USERNAME=lab4_user
DB_PASSWORD="your password"
```

8. Start the Laravel development server `php artisan serve`.

Setting up a user and creating a database

If using the SQL Shell (psql) option

Once you have installed PostgreSQL and are connected to the PostgreSQL command line interface, create a user and a database and set a password for the user and then make them the owner of the lab3 database by doing the following steps:

1. First create the user.

```sql
create user "lab4_user";
```

2. Then set a password for the user:

```sql
\password lab4_user
```

Note

The `\password lab4_user` command will prompt you to enter a password for the lab4 user. You can use whatever password you would like. Just ensure you remember it as you will need it for the next exercise.

3. Then create your database:

```sql
create database "comp3385_lab4";
```

4. Finally assign the user to be the owner of the database:

```sql
alter database comp3385_lab4 owner to lab4_user;
```

5. When you need to exit the PostgreSQL (psql) shell interface you can type:

```sql
\q
```

If using pgAdmin (GUI) option

1. Right click on **PostgreSQL 16** in the list of servers and select **Connect Server**. You will be asked to enter the password for the `postgres` admin user. Enter your the password you used during the Postgres installation and click *Ok*.

2. Right click on **Login/Group Roles** and select **Create > Login/Group Role**.

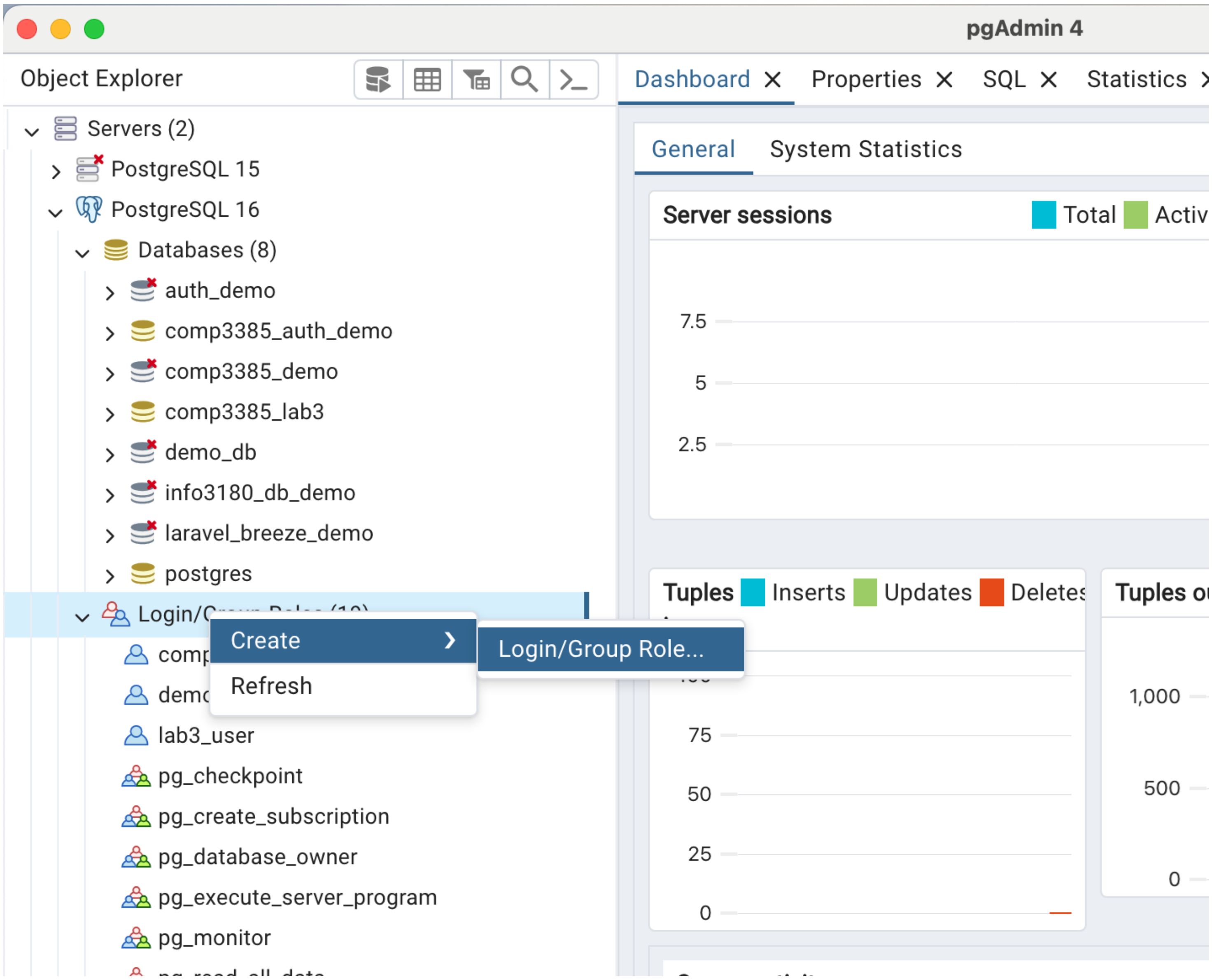


Figure 3

3. Under the **General** tab an in the Name field enter `lab4_user`. Then under the **Definition** tab enter a password for the user in the **Password** field. Lastly, under the **Privileges** tab, ensure the **Can login?** option is enabled.

4. On the list of Databases right click and select **Create > Database**.

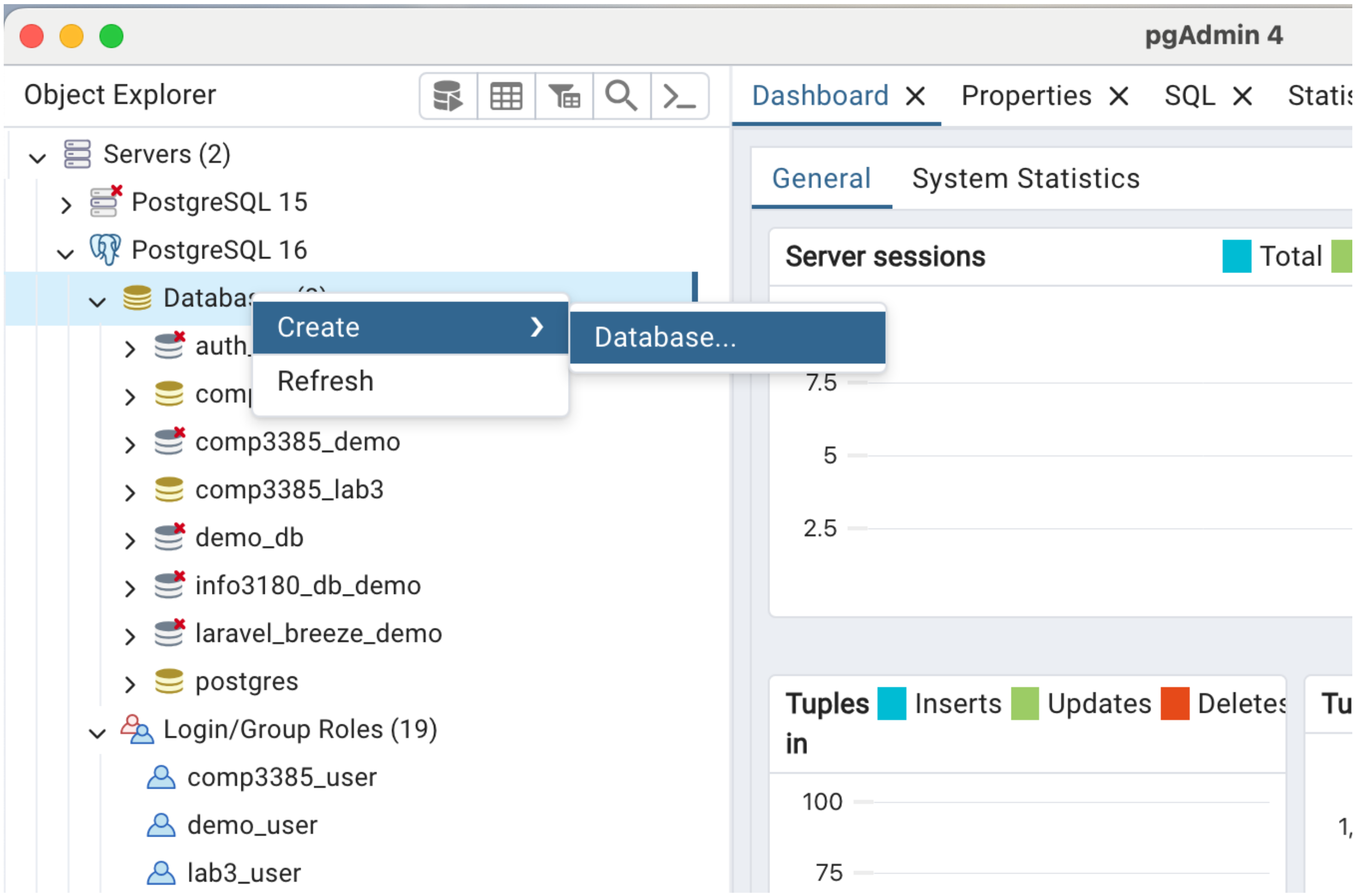


Figure 4

5. Give your database the name `comp3385_lab4`. Then in the field for **Owner**, change it to use `lab4_user` instead.

Exercise 2: Create a new Model and migration

1. Create a new Model called `Movie` to represent a `movies` table in our database using the following command:

```bash
php artisan make:model Movie
```

A new file called `Movie.php` should be created in your `app/Models` directory.

2. Create a new migration file for that new model by running the following command:

```bash
php artisan make:migration create_movies_table
```

A new file should be created in your `database/migrations` folder.

3. Open your new migration file and update it to add columns for `title`, `description`, `poster` *(just the path and filename to the uploaded file)*.

4. Run your migration to create your new database table.

```bash
php artisan migrate
```

5. Check to ensure that your database table has been created correctly.

> ⚠ **Checkpoint**
>
> Now would be a good time to commit these changes to your code to your repository and push to Github.

Exercise 3: Creating your Movie API routes

Now that your database table has been created, your next step is to create a controller called `MovieController` and API routes that your frontend will make calls to.

1. Open your `routes/api.php` file and you should create the following routes:

   - `/api/v1/movies` which is a `GET` request that maps to a `index` method in your `MovieController`
   - `/api/v1/movies` which is a `POST` request that maps to a `store` method in your `MovieController`

2. Open your `MovieController` and add the `index` and `store` methods.

3. For your `index` method you should query the movies table in your database and return all the movies in JSON format. The output should look similar to the following:

```json
{
    "message": "Movie created successfully",
    "movies": [
        {
        "id": 1,
        "title": "The movie title",
        "description": "The summary for the movie",
        "poster": "/api/v1/posters/movie-poster.jpg",
        },
        {
        "id": 2,
        "title": "Another movie title",
        "description": "The summary for the movie",
        "poster": "/api/v1/posters/movie-poster-2.jpg",
        }
    ]
}
```

4. For your `store` method you should validate the user input, save the uploaded poster image to the filesystem and then save the movie information to your movies database table. If successful, you should return a message and details about the movie saved in JSON format. The output should look similar to the following:

```json
{
    "message": "Movie created successfully",
    "movie": {
        "title": "Testing",
        "description": "Testing 123",
        "poster": "posters/the-movie-poster.png",
        "updated_at": "2024-03-31T22:13:36",
        "created_at": "2024-03-31T22:13:36",
        "id": 1
    }
}
```

If your validation fails then you should return something similar to the following JSON output:

```json
{
    "message": "The description field is required. (and 1 more error)",
    "errors": {
        "description": [
            "The description field is required."
        ],
        "poster": [
            "The poster field is required."
        ]
        }
    }
```

Hint

Laravel can actually handle sending validation error messages for you. As part of your Postman or AJAX request, ensure you send your request with the `Accept` header set to `application/json`.

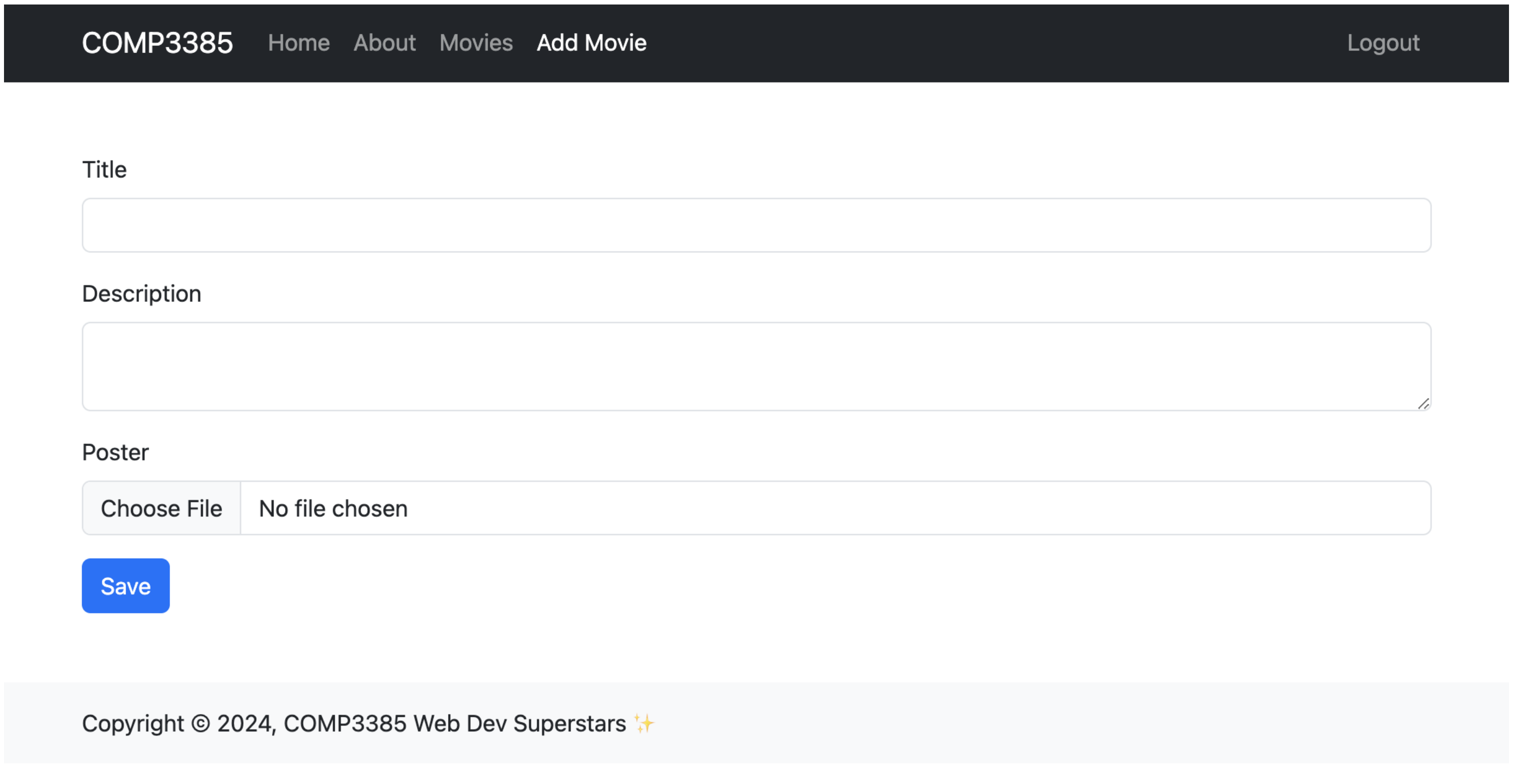
5. Test your API Endpoints using Postman (or similar API testing client).

⚠ Checkpoint

Now would be a good time to commit these changes to your code to your repository and push to Github.

Exercise 4: Create your VueJS Front-end

Now you will create a front-end for your application using VueJS that will display the form to add a new movie. When the submit button is clicked it should make an AJAX request to the API route (endpoint) you created in the previous exercise.



Copyright © 2024, COMP3385 Web Dev Superstars ✨

1. Start by first creating a new VueJS component called `MovieForm` in a file called `MovieForm.vue` in your `resources/js/components` folder. This component should have a `<template>` block that will contain the HTML code for the form.

2. On the `<form>` tag, you will use the VueJS directive `@submit.prevent="saveMovie"`. This will ensure that when the submit button is clicked or if the user presses the ENTER key on their keyboard that the function `saveMovie` (which we will define next) will be called. The `.prevent` will ensure that the default action for the form will be prevented. This is similar to the `preventDefault()` function that you used in COMP2245.

3. Within your `<form></form>` tags you will need to create your form label and input tags for each field. For example:

   ```html
   <div class="form-group mb-3">
   <label for="title" class="form-label">Movie Title</label>
   <input type="text" name="title" class="form-control" />
   </div>
   ```

4. Next, in the `<script setup>` block of your component define the function called `saveMovie`. This will be responsible for making the AJAX request using the Fetch API to your API endpoint `/api/v1/movies` that you created in your `routes/api.php` file. Start with the following example for your `fetch()` function:

   ```js
   fetch("/api/v1/movies", {
       method: 'POST',
       headers: {
           'Accept': 'application/json'
       }
   })
   .then(function (response) {
       return response.json();
   })
   .then(function (data) {
       // display a success message
       console.log(data);
   })
   .catch(function (error) {
   ```

```
        console.log(error);
    });
```

5. Now we will now need to create a route and a page to display this component in our Frontend.

   Create another component called `AddMovieView.vue` in your `resources/js/Pages` folder and import your `MovieForm` component that you previously created and use it in the `<template>` block for `AddMovieView.vue`. You also need to ensure you update your `resources/js/router/index.js` file to add the route `/movies/create` to the VueRouter.

6. Now start your Vite dev server by running:

   ```bash
   npm run dev
   ```

   Visit your newly created VueJS route, http://localhost:8000/movies/create . Assuming you did everything correctly, try to submit the form without anything in your form fields and look in your Web Browser Developer Tools Console. You should get a response that lists your form validation errors in the console of your web browser. Take note of what these errors are. Now try actually adding a description and uploading a file. Do you still get any errors? What error do you see?

7. We haven't actually sent any data along with our AJAX request. Since we are sending a file along with our request we will take advantage of the `FormData` interface that is available to us in JavaScript. The `FormData` interface provides a way to easily construct a set of key/value pairs representing form fields and their values, which can then be easily sent as part of our AJAX request. It uses the same format a form would use if the encoding type were set to `multipart/form-data`. Which is what we need when sending a file. Update your `saveMovie` method in your component to have the following:

   ```js
   let movieForm = document.getElementById('movieForm');
   let form_data = new FormData(movieForm);

   fetch("/api/v1/movies", {
       method: 'POST',
       body: form_data,
       headers: {
           'Accept': 'application/json'
       }
   })
   ```

```
.then(function (response) {
    return response.json();
})
.then(function (data) {
    // display a success message
    console.log(data);
})
.catch(function (error) {
    console.log(error);
});
```

Note

You will also need to add an `id` attribute with a value of `movieForm` to your `<form>` tag so that we can specifically reference that form in our `FormData` interface.

8. Now fill out your form again and submit the form, what do you see in your browser console?

9. Next, ensure that you add a link in the navigation bar in your `Header` component for the new route you created for the add movie form page in VueJS.

Bonus

Now see if you can figure out how to give the user feedback by displaying the success or error message on the same movie form page (instead of in the console) when you have a successful upload or the validation fails.

Hint

You will need to define some reactive state properties within your component and possibly use the `v-if` and `v-for` VueJS directives in your template to hide/show the message. You will also need to tweak one of the `then()` methods in your `fetch()` AJAX request to determine whether to display the *success* or *error* messages.

⚠ Checkpoint

Now would be a good time to commit these changes to your code to your repository and push to Github.

10. Now that we are able to add movies to our database via our VueJS Frontend and our Laravel API, next we want to display the list of our movies on a page.

Create a new VueJS page called `MoviesView.vue` in your `resources/js/Pages` folder. In your `<script setup></script>` tags, import your `ref` and `onMounted` functions from vue and create a reactive property called `movies` which is an empty array.

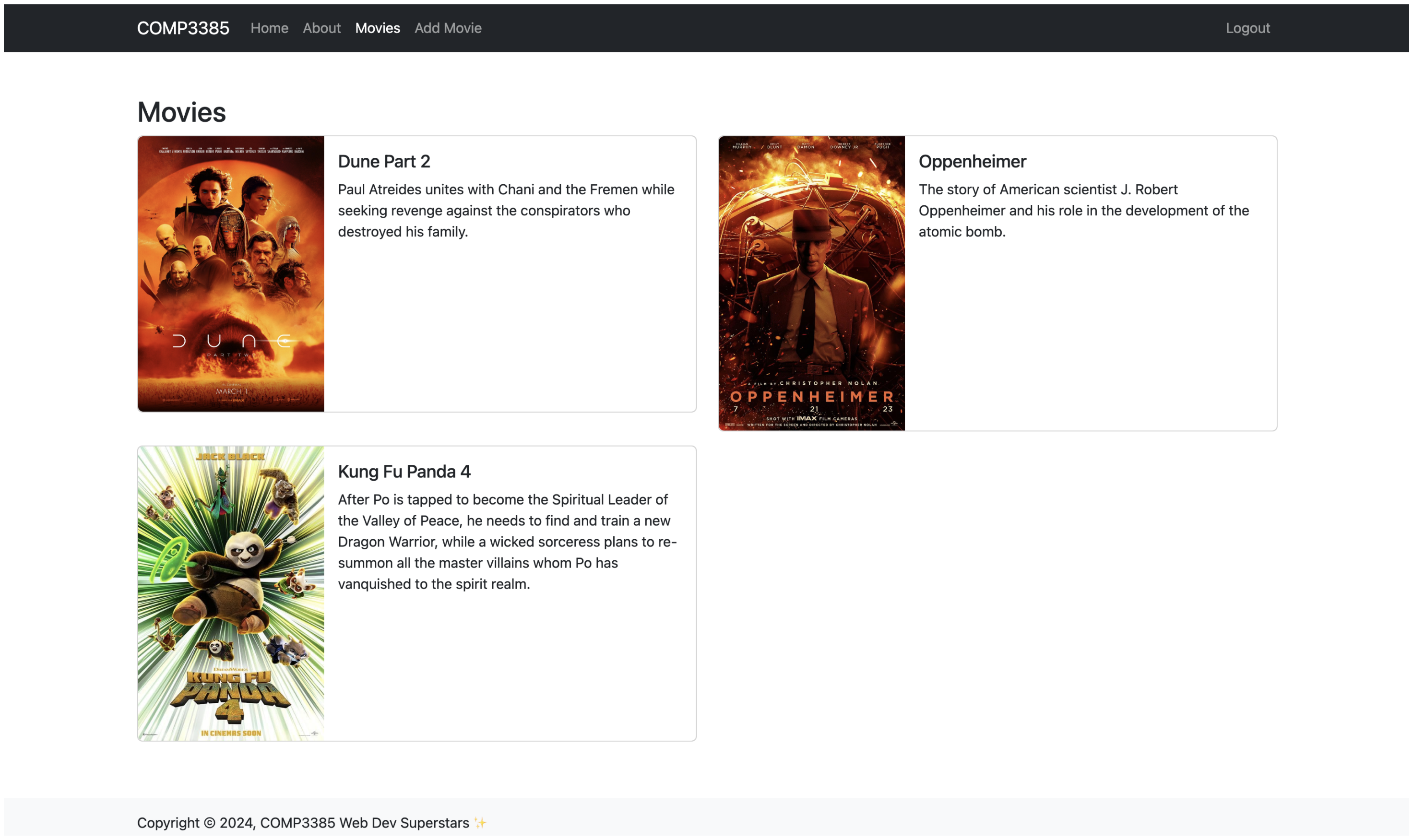
```vue
<script setup>
import { ref, onMounted } from "vue";

let movies = ref([]);
</script>
```

11. Now add a function called `fetchMovies()` in your `<script setup></script>` tags that will make an AJAX request to your Laravel API endpoint `/api/v1/movies` as a `GET` request this time. You will run that function in your `onMounted` life cycle hook. And ensure that you update your `movies` reactive property with the data returned from the API call.

12. Next, you will update your `<template></template>` for your `MoviesView` page component with the necessary HTML and VueJS directives to loop over the list of movies and display the *poster*, *movie title* and *movie description* in a card.

13. Add a route `/movies` to your vue-router (in `resources/js/router/index.js`) that will load your `MoviesView` page and add another router link to the navigation bar in your `Header` component for the new route you created for the page to view all the movies in VueJS. Your final page should look like the following:


COMP3385 Home About Movies Add Movie Logout

Movies

Dune Part 2

Paul Atreides unites with Chani and the Fremen while seeking revenge against the conspirators who destroyed his family.

Oppenheimer

The story of American scientist J. Robert Oppenheimer and his role in the development of the atomic bomb.

Kung Fu Panda 4

After Po is tapped to become the Spiritual Leader of the Valley of Peace, he needs to find and train a new Dragon Warrior, while a wicked sorceress plans to re-summon all the master villains whom Po has vanquished to the spirit realm.

Copyright © 2024, COMP3385 Web Dev Superstars ✨

Note

Displaying the image might be a little tricky. Instead of just using the `src` attribute by itself on an `<img />` tag, try using `:src`. This is shorthand for `v-bind:src` which is another VueJS directive that allows us to bind a string to the attribute.

If we were to deploy this VueJS/Laravel API to a production server there would be a few extra changes to make but we will leave it like this for now.

Exercise 5: Use a JWT to secure your API

In this part of the assignment we will now secure out API using a JSON Web Token (JWT). To do this we will need to pull on a third-party package called `PHP-Open-Source-Saver/jwt-auth`.

1. To install the package, run the following command in your project:

```bash
composer require php-open-source-saver/jwt-auth
```

2. Next run:

```bash
php artisan vendor:publish --provider="PHPOpenSourceSaver\JWTAuth\Providers\LaravelS
```

You should now have a `config/jwt.php` file that allows you configure the basics of this package.

3. There is also a helper command that can generate a `SECRET` for use in signing your JWT's:

```bash
php artisan jwt:secret
```

4. Now open your `app\Models\User.php` file and update it with the following highlighted lines:

```php
<?php

namespace App\Models;

// use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use PHPOpenSourceSaver\JWTAuth\Contracts\JWTSubject;

class User extends Authenticatable implements JWTSubject
{
    use HasFactory, Notifiable;

    // Rest omitted for brevity

    /**
     * Get the identifier that will be stored in the subject claim of the JWT.
     *
     * @return mixed
     */
    public function getJWTIdentifier()
    {
        return $this->getKey();
    }

    /**
     * Return a key value array, containing any custom claims to be added to the JWT
```

```
         *
         * @return array
         */
        public function getJWTCustomClaims()
        {
            return [];
        }
    }
```

5. Inside the `config/auth.php` file you will need to make a few changes to configure Laravel to use the `jwt` guard to power your application authentication.

   Make the following changes to the file:

   ```php
   'defaults' => [
       'guard' => 'api',
       'passwords' => 'users',
   ],

   ...

   'guards' => [
       'api' => [
           'driver' => 'jwt',
           'provider' => 'users',
       ],
   ],
   ```

   Here we are telling the `api` guard to use the `jwt` driver, and we are setting the `api` guard as the default.

6. Now we will add API routes to login and logout. Open `routes/api.php` and add:

   ```php
   Route::post('/v1/login', [AuthController::class, 'login'])
   Route::post('/v1/logout', [AuthController::class, 'logout'])
   ```

7. Next, create the `AuthController`, either manually or by running the artisan command:

bash
```bash
php artisan make:controller AuthController
```

Then add the following:

php
```php
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Auth;
use App\Http\Controllers\Controller;

class AuthController extends Controller
{
    /**
     * Get a JWT via given credentials.
     *
     * @return \Illuminate\Http\JsonResponse
     */
    public function login()
    {
        $credentials = request(['email', 'password']);

        if (! $token = auth()->attempt($credentials)) {
            return response()->json(['error' => 'Unauthorized'], 401);
        }

        return response()->json([
            'message' => 'Login Successful!',
            'access_token' => $token,
        ]);
    }

    /**
     * Log the user out (Invalidate the token).
     *
     * @return \Illuminate\Http\JsonResponse
     */
    public function logout()
    {
        auth()->logout();
```

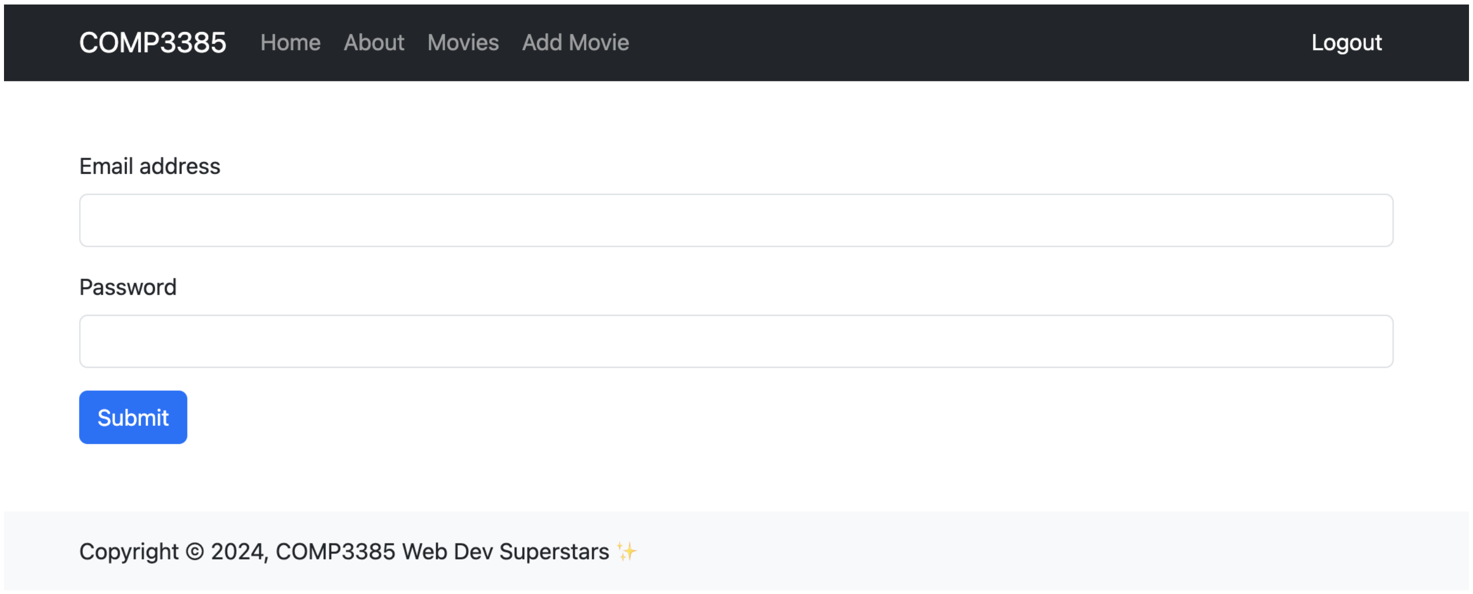
```
            return response()->json(['message' => 'Successfully logged out']);
        }
    }
```

8. Run the database seeder to add a user to your users table:

```bash
php artisan db:seed
```

> **Note**
>
> This should add a user with the email `test@example.com` and password as `password`. You can look at the `database/factories/UserFactory.php` and `database/seeders/DatabaseSeeder.php` files to see who this is done.

9. Now you will need to create a `/login` route in your VueJS router and a page called `LoginView.vue` to go along with it. Create a login form that when submitted makes an AJAX request to your `api/v1/login` endpoint that if successful will return a JWT access token that you will store on the client-side and then send along with any future AJAX requests to your other secure API endpoints.



Copyright © 2024, COMP3385 Web Dev Superstars ✨

> **Hint**
>
> You can use either localStorage or a Cookie to store your JWT.

10. Update your Movie API endpoints to use your `auth:api` middleware. This will check to ensure that your JWT has been sent along with the request and that it is a valid JWT before allowing the request to be handled by the controller.

    Hint

    Add `->middleware('auth:api')` to the end of your movie route functions in `routes/api.php`.

11. Ensure that you now update your AJAX requests to your movies API endpoints in your VueJS frontend to send your API token as part of the `Authorization` header using the `Bearer` schema. For example:

    ```js
    fetch('/api/v1/movies', {
        'headers': {
            'Authorization': `Bearer your-jwt-access-token`
        }
    })
    ```

12. Lastly, update your VueJS `Header` component to add links to **Login** or **Logout** of the application.

    Note

    You will need to make an AJAX request when the Logout link is clicked to send a request to your `/api/v1/logout` API endpoint to log the user out of the application.

⚠ **Checkpoint**

Now would be a good time to commit these changes to your code to your repository and push to Github.

Submission

Submit your code via the "Assignment 4" link on the VLE. You should submit the following links:

- Github repository URL for your Laravel Exercise e.g. https://github.com/{yourusername}/comp3385-assignment-4