S2T
Technical Documentation

S2T_Triple_Vision

Members:

Jacques Cloete – 44214987
Jean-Luc Begue – 40779173
Danika le Roux – 41049764

Contents

Front-End Architecture 2
ENTERING THE BACK-END: 3
2. Database Schema: 3
3. Improved Database Schema: 4
5. Migrations: 5
6. Models: 6
7. Seeders: 7
8. Controllers: 8
ENTERING THE FRONTEND: 13
12. Routes: 13
Web.php 13
Api.php 14
..... 15
17. Conclusion: 17

1. Why Did We Select Vue.js & Laravel?

Back-End Architecture

Our Laravel + Vue.js + Vuetify web application's backend architecture uses the Model-View-Controller (MVC) pattern, which divides the application functionality into three interdependent parts. The foundation for developing the API, responding to queries, handling data management using Eloquent ORM, and putting business logic into practice is provided by Laravel. Incoming requests are handled by the controllers, which also communicate with models to manipulate data and provide JSON responses to the frontend. In order to handle routes and middleware for authentication, authorization, and error management efficiently, this structure encourages clean code organization and maintainability. Additionally, by adhering to RESTful API principles, the backend is guaranteed to be scalable and able to support numerous client applications.

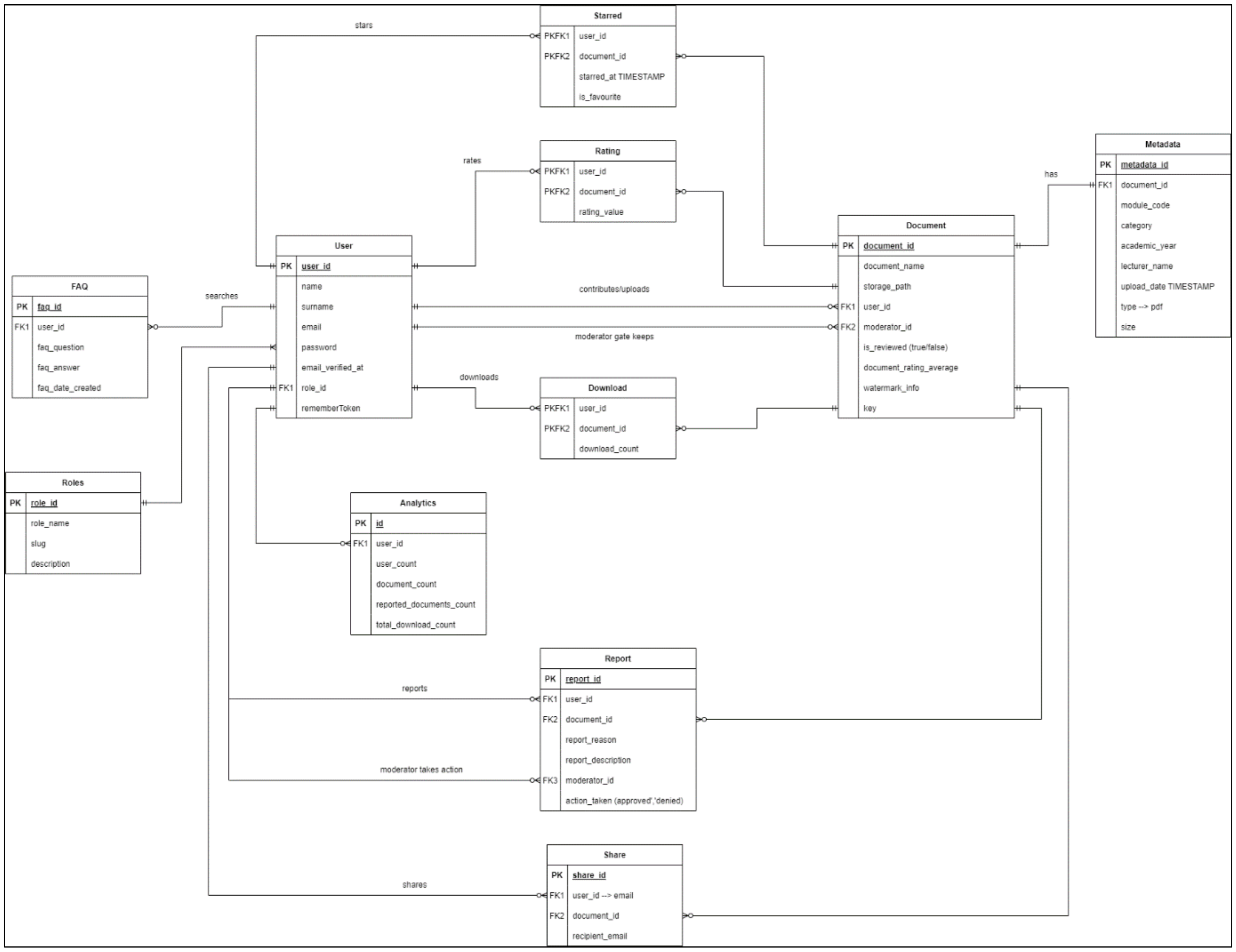
Front-End Architecture

The frontend architecture makes use of Vuetify and Vue.js, two progressive JavaScript frameworks, to produce a vibrant, responsive user interface. Because of the component-based design employed by this architecture, it is possible to create reusable user interface components that contain their own logic and state. To improve the Single Page Application (SPA) experience, Vue Router is used to manage navigation between several views. Vuex handles state management, allowing you centralized control over the state of the application and easing data sharing between components. The frontend and Laravel backend communicate effortlessly thanks to the integration of Axios for API queries, which guarantees effective data processing and fetching. The dynamic, user-friendly interface that this design produces improves application speed and user engagement.

ENTERING THE BACK-END:

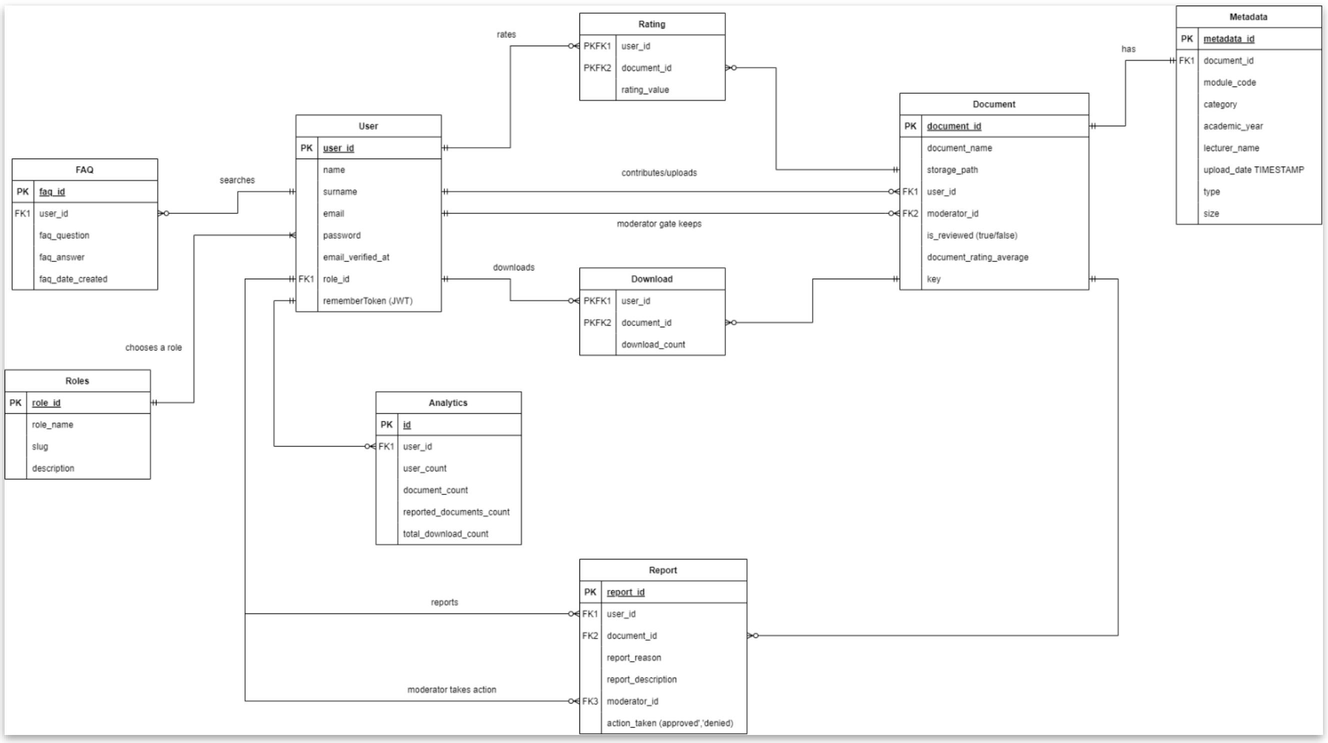
2. Database Schema:

We reviewed the **S2TTech&FuncSpec - 323 v1.1** document and subsequently created a database schema that outlines the structure and relationships of our database entities. This schema serves as a blueprint for the database design.



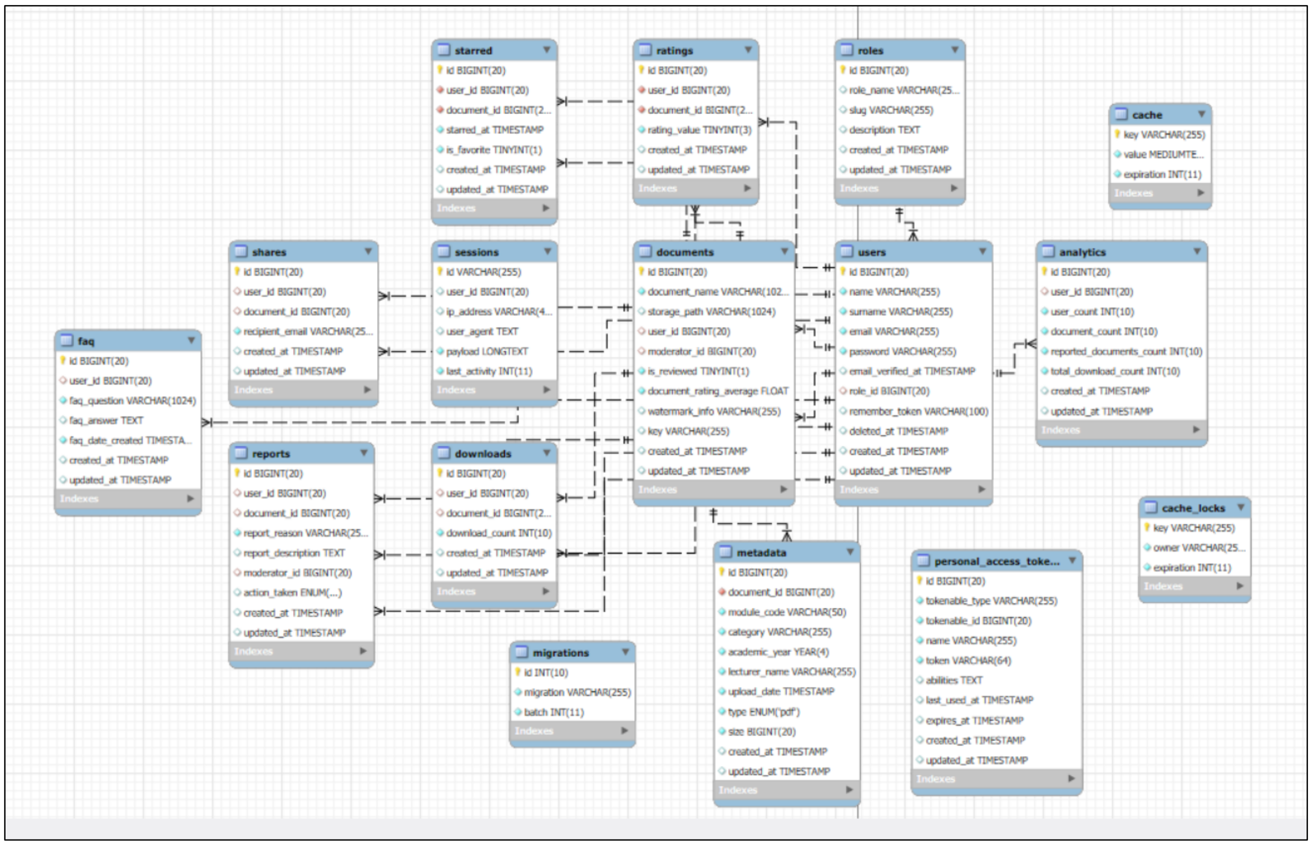
- https://tinyurl.com/5xedj45e

3. Improved Database Schema:



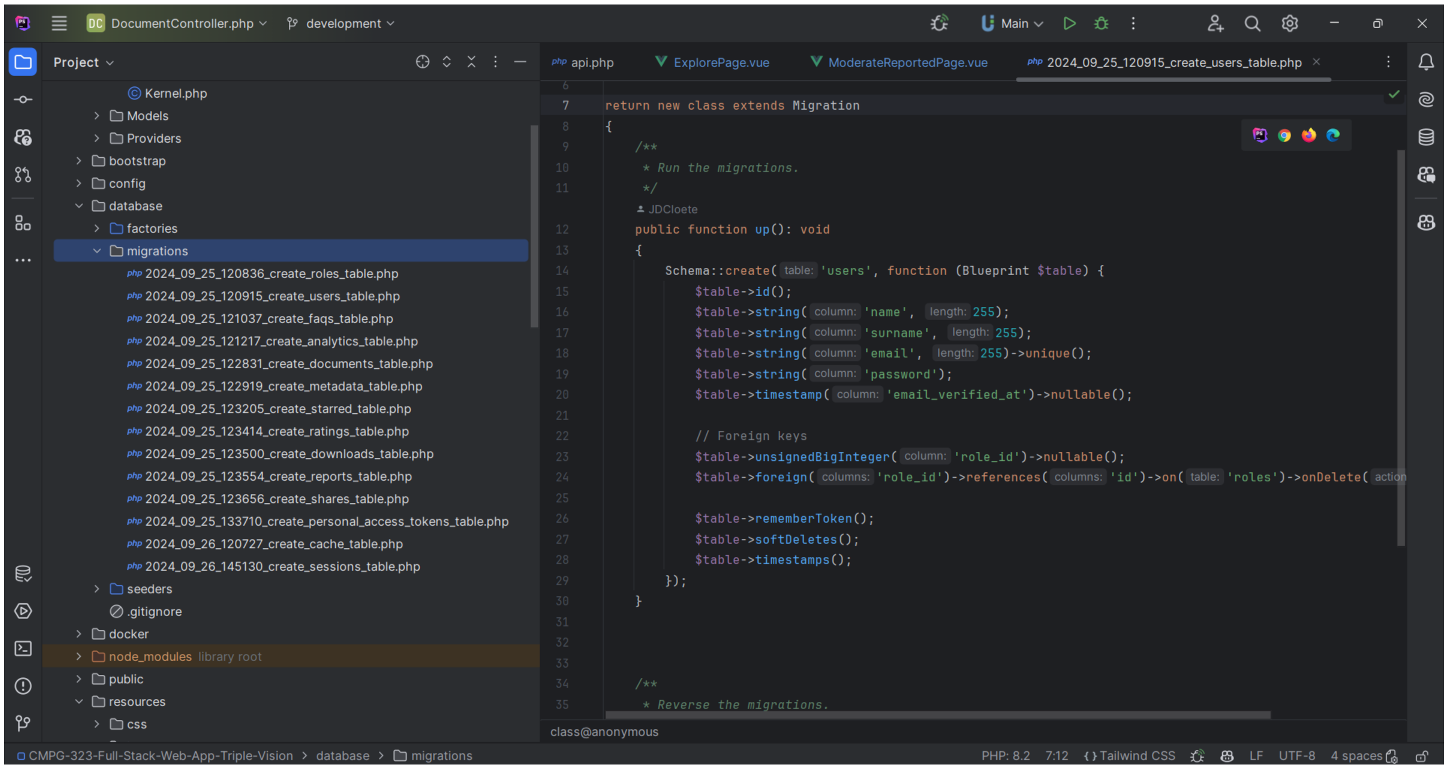
https://tinyurl.com/bdesymed

4. MySQL ERD



5. Migrations:

Migrations are a way to version control database changes in Laravel. They allow developers to define the database structure in PHP code, which can be easily shared and deployed. Migrations ensure that the database schema is in sync across different environments and allow for easy modifications and rollbacks.



```
Kernel.php
> Models
> Providers
> bootstrap
> config
v database
  > factories
  v migrations
      2024_09_25_120836_create_roles_table.php
      2024_09_25_120915_create_users_table.php
      2024_09_25_121037_create_faqs_table.php
      2024_09_25_121217_create_analytics_table.php
      2024_09_25_122831_create_documents_table.php
      2024_09_25_122919_create_metadata_table.php
      2024_09_25_123205_create_starred_table.php
      2024_09_25_123414_create_ratings_table.php
      2024_09_25_123500_create_downloads_table.php
      2024_09_25_123554_create_reports_table.php
      2024_09_25_123656_create_shares_table.php
      2024_09_25_133710_create_personal_access_tokens_table.php
      2024_09_26_120727_create_cache_table.php
      2024_09_26_145130_create_sessions_table.php
  > seeders
    .gitignore
> docker
> node_modules library root
> public
v resources
  > css
```

```php
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    JDCloete
    public function up(): void
    {
        Schema::create( table: 'users', function (Blueprint $table) {
            $table->id();
            $table->string( column: 'name',  length: 255);
            $table->string( column: 'surname',  length: 255);
            $table->string( column: 'email',  length: 255)->unique();
            $table->string( column: 'password');
            $table->timestamp( column: 'email_verified_at')->nullable();

            // Foreign keys
            $table->unsignedBigInteger( column: 'role_id')->nullable();
            $table->foreign( columns: 'role_id')->references( columns: 'id')->on( table: 'roles')->onDelete( action

            $table->rememberToken();
            $table->softDeletes();
            $table->timestamps();
        });
    }


    /**
     * Reverse the migrations.
```

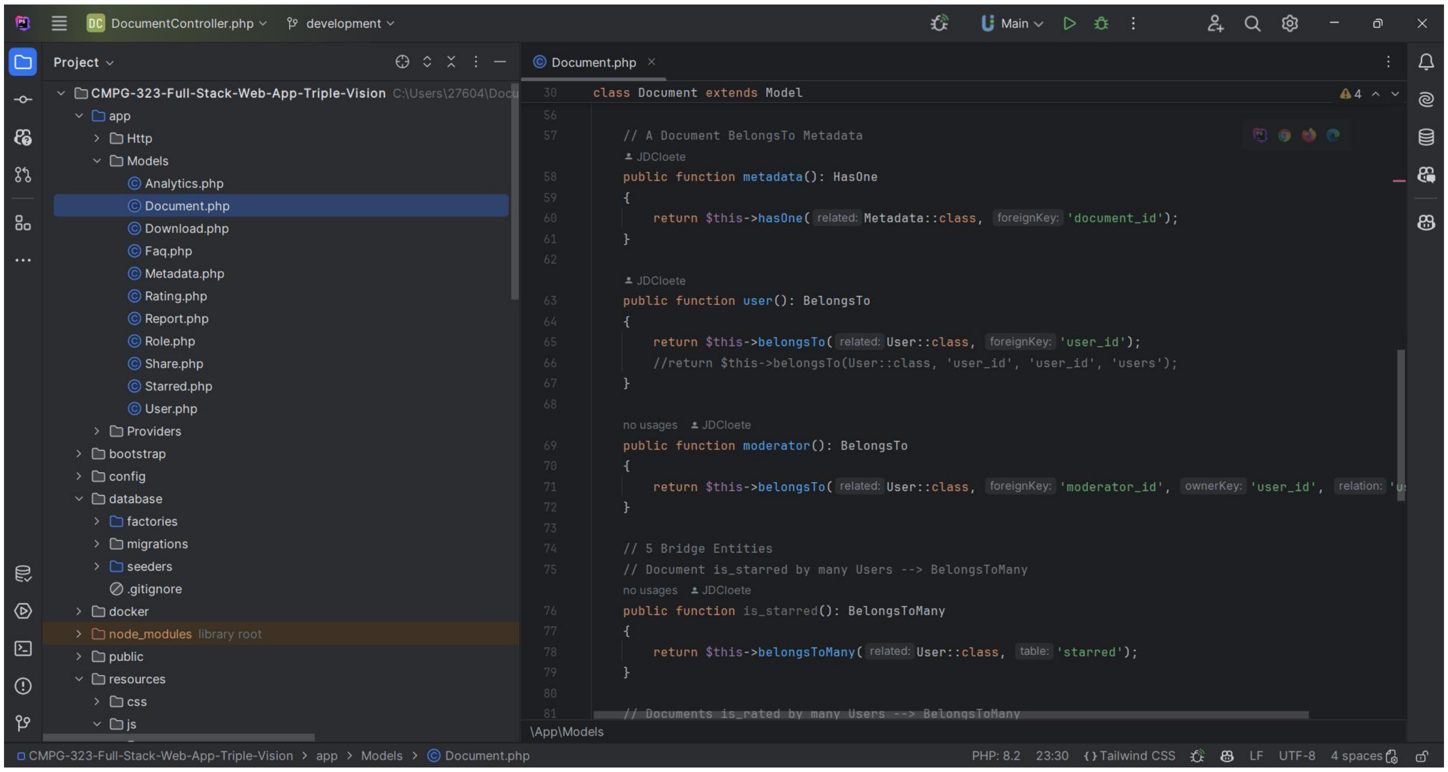
```
class@anonymous
CMPG-323-Full-Stack-Web-App-Triple-Vision > database > migrations          PHP: 8.2   7:12   Tailwind CSS   LF   UTF-8   4 spaces
```

In the above screenshot we can see that this Laravel migration script creates a user's table with fields for storing user details such as name, surname, email (which is unique), and password. The email_verified_at field tracks email verification, and rememberToken supports "remember me" functionality for login sessions.

The migration also defines a foreign key role_id, which references the roles table, allowing each user to have an associated role. If a role is deleted, the role_id is set to null. The softDeletes feature enables soft deletion of users, keeping the data for potential restoration. Timestamps (created_at and updated_at) are automatically managed. The down method reverses these changes by dropping the table.

6. Models:

Models in Laravel represent the entities in our application and interact with the corresponding database tables. Each model encapsulates the properties and behaviours of a particular entity, following the ActiveRecord pattern.
By utilizing models, we can easily manage data retrieval, insertion, and manipulation, abstracting complex SQL queries into simple method calls.



```php
class Document extends Model

    // A Document BelongsTo Metadata
    JDCloete
    public function metadata(): HasOne
    {
        return $this->hasOne( related: Metadata::class,  foreignKey: 'document_id');
    }

    JDCloete
    public function user(): BelongsTo
    {
        return $this->belongsTo( related: User::class,  foreignKey: 'user_id');
        //return $this->belongsTo(User::class, 'user_id', 'user_id', 'users');
    }

    no usages  JDCloete
    public function moderator(): BelongsTo
    {
        return $this->belongsTo( related: User::class,  foreignKey: 'moderator_id',  ownerKey: 'user_id',  relation: 'u
    }

    // 5 Bridge Entities
    // Document is_starred by many Users --> BelongsToMany
    no usages  JDCloete
    public function is_starred(): BelongsToMany
    {
        return $this->belongsToMany( related: User::class,  table: 'starred');
    }

    // Documents is_rated by many Users --> BelongsToMany
```

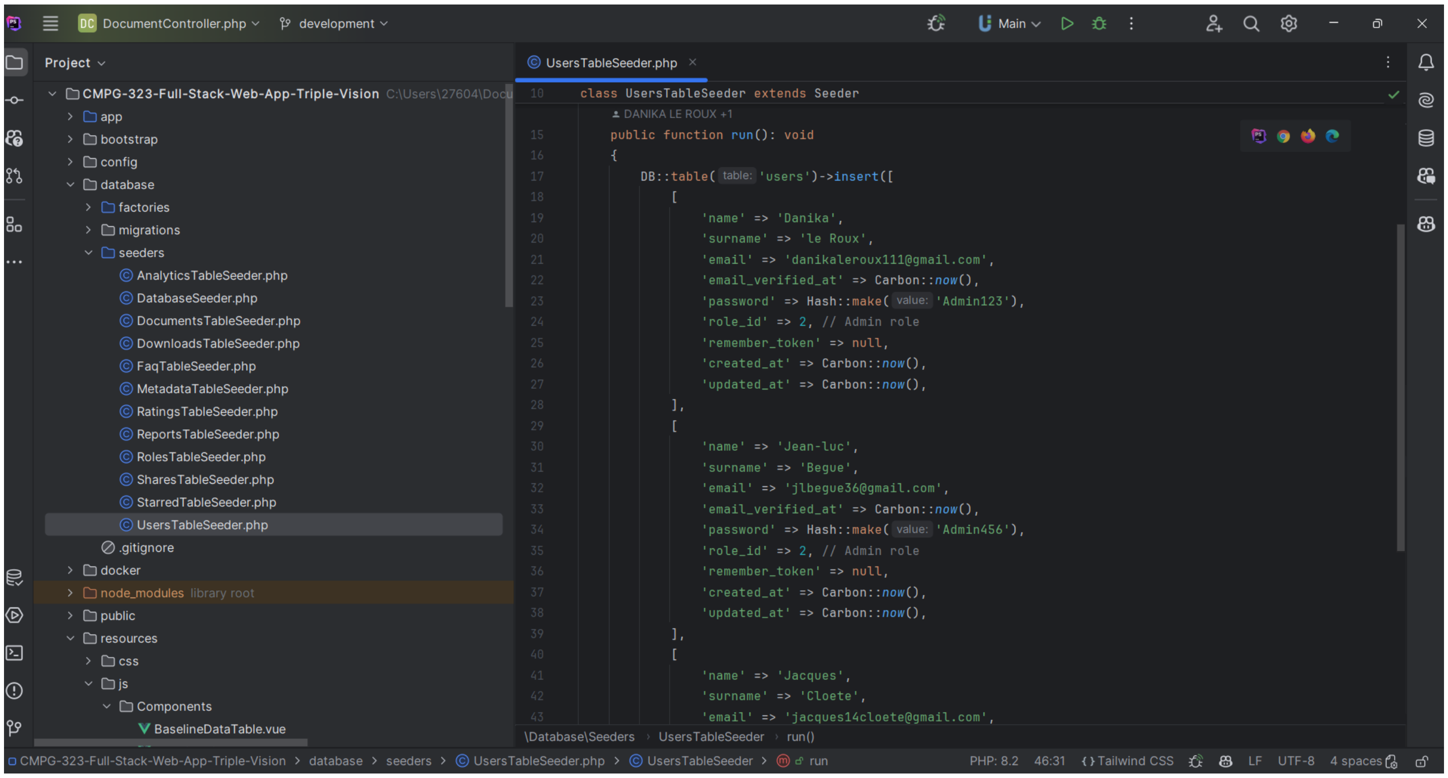
The Document model in Laravel is used to define the structure and relationships of documents in your application. It specifies attributes such as document_name, storage_path, user_id, moderator_id, and document_rating_average, while using Laravel's HasFactory and Notifiable traits. The model defines several key relationships, including:

- **BelongsTo** relationships with the User model (both as the creator of the document and as a moderator).
- **HasOne** relationship with Metadata, indicating a document has associated metadata.
- **BelongsToMany** relationships with the User model for actions like starring, rating, downloading, reporting, and sharing documents, indicating many-to-many relationships through pivot tables.
- It also includes attributes casting for types such as integers, booleans, and floats. These relations make this model central to how users interact with documents, allowing for features like user reviews, moderation, and document sharing.

Using PhpStorm with Laravel, you gain features like auto-completion, error checking, and navigation, making it easier to manage such models and their relationships effectively.

7. Seeders:

Seeders are classes that enable us to populate our database with sample or initial data. They help in testing the application and ensure that the database has the necessary data to function correctly. By defining seeder classes, we can easily create and run data population scripts that enhance the development and testing processes.



```
CMPG-323-Full-Stack-Web-App-Triple-Vision C:\Users\27604\Docu
  app
  bootstrap
  config
  database
    factories
    migrations
    seeders
      AnalyticsTableSeeder.php
      DatabaseSeeder.php
      DocumentsTableSeeder.php
      DownloadsTableSeeder.php
      FaqTableSeeder.php
      MetadataTableSeeder.php
      RatingsTableSeeder.php
      ReportsTableSeeder.php
      RolesTableSeeder.php
      SharesTableSeeder.php
      StarredTableSeeder.php
      UsersTableSeeder.php
    .gitignore
  docker
  node_modules library root
  public
  resources
    css
    js
      Components
        BaselineDataTable.vue
```

```php
class UsersTableSeeder extends Seeder
    DANIKA LE ROUX +1
    public function run(): void
    {
        DB::table( table: 'users')->insert([
            [
                'name' => 'Danika',
                'surname' => 'le Roux',
                'email' => 'danikaleroux111@gmail.com',
                'email_verified_at' => Carbon::now(),
                'password' => Hash::make( value: 'Admin123'),
                'role_id' => 2, // Admin role
                'remember_token' => null,
                'created_at' => Carbon::now(),
                'updated_at' => Carbon::now(),
            ],
            [
                'name' => 'Jean-luc',
                'surname' => 'Begue',
                'email' => 'jlbegue36@gmail.com',
                'email_verified_at' => Carbon::now(),
                'password' => Hash::make( value: 'Admin456'),
                'role_id' => 2, // Admin role
                'remember_token' => null,
                'created_at' => Carbon::now(),
                'updated_at' => Carbon::now(),
            ],
            [
                'name' => 'Jacques',
                'surname' => 'Cloete',
                'email' => 'jacques14cloete@gmail.com',
```

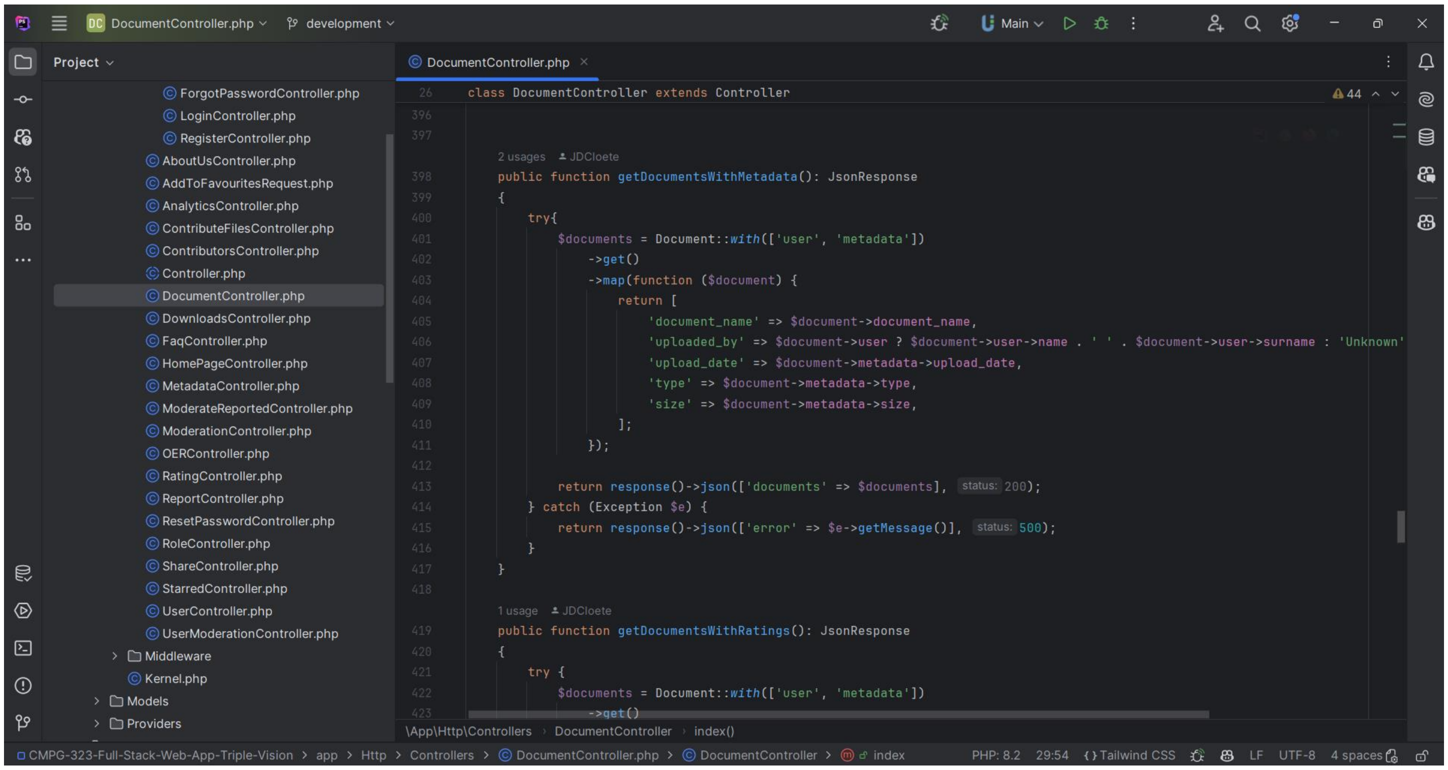
```
\Database\Seeders > UsersTableSeeder > run()
CMPG-323-Full-Stack-Web-App-Triple-Vision > database > seeders > UsersTableSeeder.php > UsersTableSeeder > run    PHP: 8.2  46:31  Tailwind CSS  LF  UTF-8  4 spaces
```

This **UsersTableSeeder** in Laravel is responsible for populating the users table with predefined data. It inserts three users, each with attributes such as name, surname, email, hashed passwords (using Hash::make), and associated roles (role_id). Timestamps for created_at and updated_at are set using **Carbon**, ensuring accurate date/time data. The email_verified_at field marks the email as verified at the current time.

This seeder is typically used during development to quickly populate the database with sample users, aiding testing and application setup.

8. Controllers:

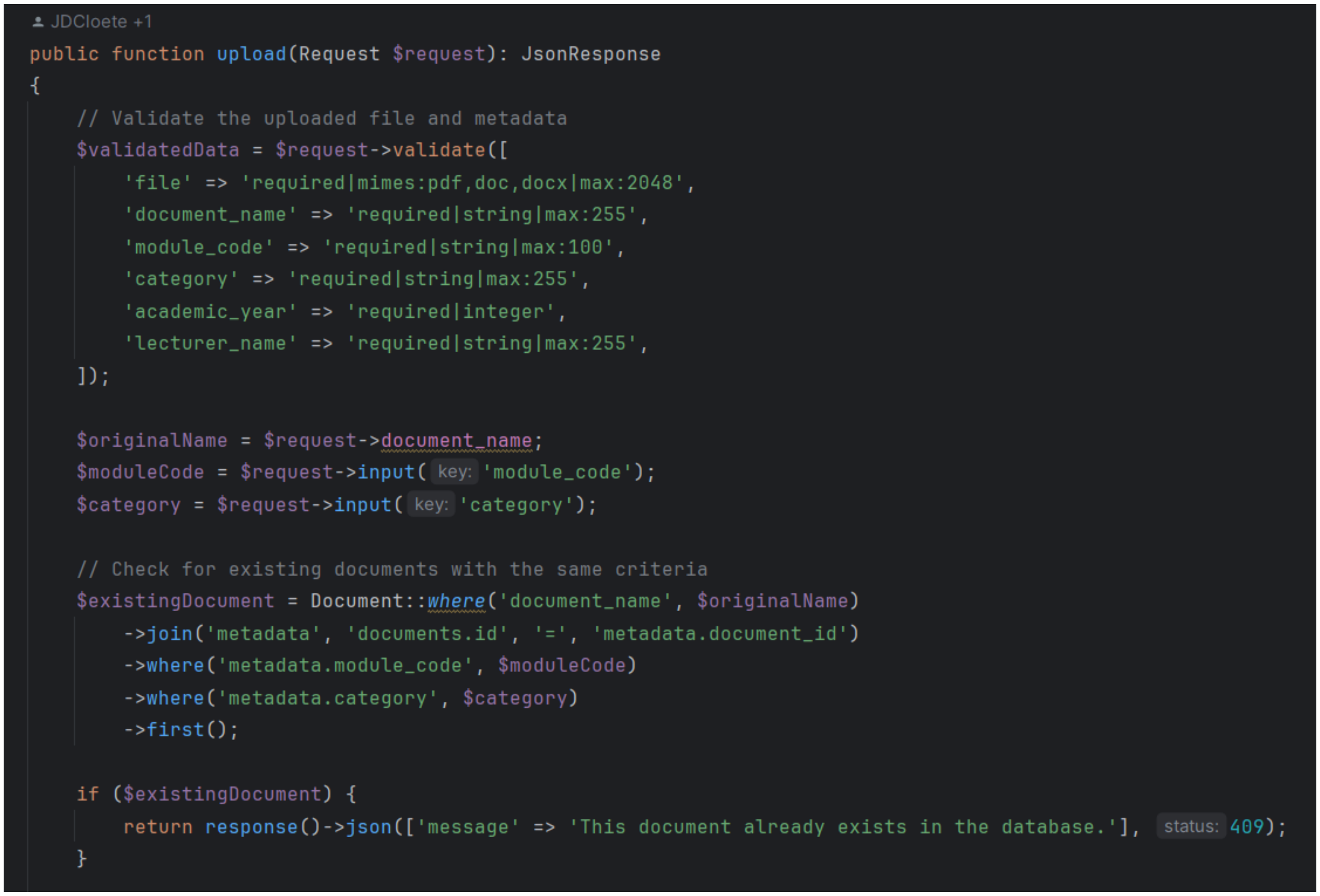
Controllers are responsible for handling incoming requests, processing user input, and returning appropriate responses. In our application, controllers serve as intermediaries between the models and views, encapsulating the business logic. Each controller is tailored to manage specific functionalities, making the application modular and organized.



```php
class DocumentController extends Controller


    2 usages  JDCloete
    public function getDocumentsWithMetadata(): JsonResponse
    {
        try{
            $documents = Document::with(['user', 'metadata'])
                ->get()
                ->map(function ($document) {
                    return [
                        'document_name' => $document->document_name,
                        'uploaded_by' => $document->user ? $document->user->name . ' ' . $document->user->surname : 'Unknown'
                        'upload_date' => $document->metadata->upload_date,
                        'type' => $document->metadata->type,
                        'size' => $document->metadata->size,
                    ];
                });

            return response()->json(['documents' => $documents], status: 200);
        } catch (Exception $e) {
            return response()->json(['error' => $e->getMessage()], status: 500);
        }
    }

    1 usage  JDCloete
    public function getDocumentsWithRatings(): JsonResponse
    {
        try {
            $documents = Document::with(['user', 'metadata'])
                ->get()
```

This Laravel route and controller logic facilitates the population of a moderation page with documents, using Eloquent to retrieve data from the database. The getDocumentsWithMetadata method fetches documents along with associated user and metadata information, such as upload date, type, and size. It ensures proper handling of missing users, displaying "Unknown" if the user is not found. The index method filters documents to display only those that have been reviewed and returns the data to the front-end using Inertia.js. This setup allows efficient data management and real-time interaction, with a focus on clean code architecture for document moderation. PhpStorm's deep Laravel integration can help auto-generate routes, assist with code navigation, and debug potential issues effectively during the development of this feature.

9. Explanation of the Upload function and Processing Function that validates, Converts and Watermarks documents uploaded.



```
JDCloete +1
public function upload(Request $request): JsonResponse
{
    // Validate the uploaded file and metadata
    $validatedData = $request->validate([
        'file' => 'required|mimes:pdf,doc,docx|max:2048',
        'document_name' => 'required|string|max:255',
        'module_code' => 'required|string|max:100',
        'category' => 'required|string|max:255',
        'academic_year' => 'required|integer',
        'lecturer_name' => 'required|string|max:255',
    ]);

    $originalName = $request->document_name;
    $moduleCode = $request->input( key: 'module_code');
    $category = $request->input( key: 'category');

    // Check for existing documents with the same criteria
    $existingDocument = Document::where('document_name', $originalName)
        ->join('metadata', 'documents.id', '=', 'metadata.document_id')
        ->where('metadata.module_code', $moduleCode)
        ->where('metadata.category', $category)
        ->first();

    if ($existingDocument) {
        return response()->json(['message' => 'This document already exists in the database.'], status: 409);
    }
```

This **upload function** is the pride and joy of the website as it handles document uploads with validation, file conversions, watermarking, and storage. Here's a breakdown of each section:

---

1. Validation

- Validates the uploaded file and related metadata to ensure proper file types, document name, module code, category, academic year, and lecturer information.

2. Duplicate Check

- Searches the database for existing documents with the same name, module code, and category to prevent duplicate uploads.

3. File Upload Handling

- Extracts file details such as extension and size. Handles custom file paths for document storage.

4. DOCX to PDF Conversion

- Converts .docx files to .pdf using PHPWord and DomPDF if necessary, ensuring uniformity in document handling.

5. Watermarking & PDF Merging

- If the file is a PDF (or converted to PDF), adds a cover page, merges the uploaded document, and applies a watermark on each page.

6. File Storage

- Stores the final file (watermarked/converted) in a predefined folder structure. For non-PDF files, it bypasses watermarking.

7. Database Insertion

- Uses a database transaction to store document and metadata information in two related tables: documents and metadata.

8. Error Handling

- Catches exceptions during file processing and database insertion to provide meaningful error messages.

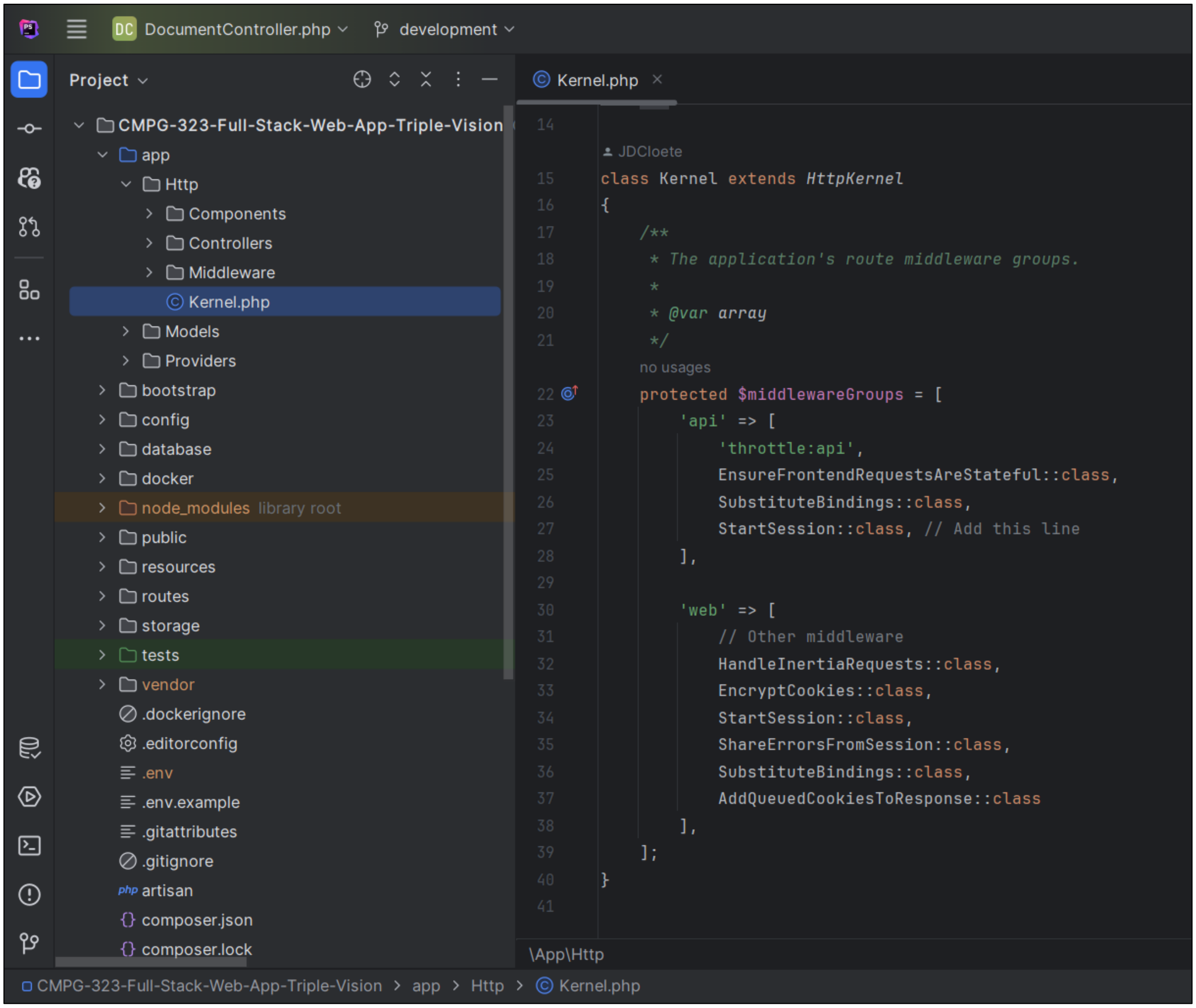
9. Success Response

- Returns a success message after the file has been validated, converted (if necessary), watermarked, and stored successfully.

This structured approach ensures smooth document uploads, proper validation, and data integrity, all while preventing duplicates and adding value with watermarks and file conversions.

10. Middleware Configuration in Kernel.php for Web and API Routes

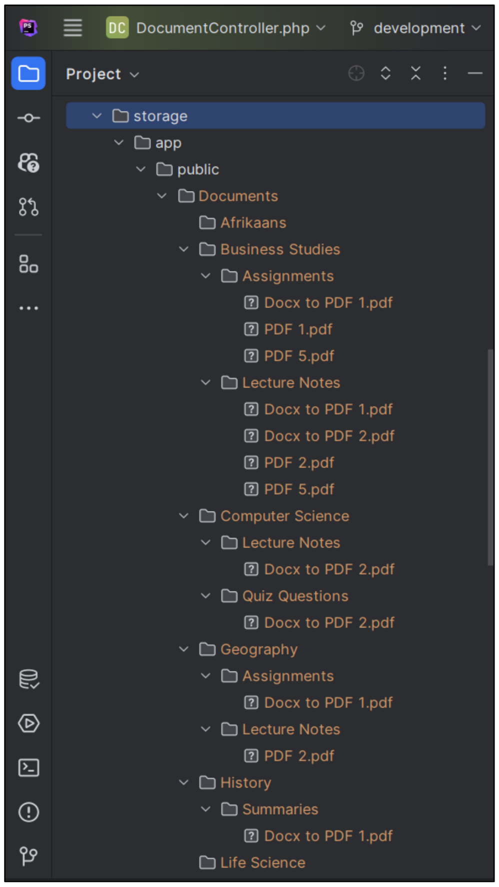
This **Kernel.php** file plays a crucial role in a Laravel application by managing middleware. Middleware is responsible for filtering HTTP requests before they reach controllers and handling HTTP responses after they've been processed. Here's a breakdown of the code and its purpose:



```php
class Kernel extends HttpKernel
{
    /**
     * The application's route middleware groups.
     *
     * @var array
     */
    protected $middlewareGroups = [
        'api' => [
            'throttle:api',
            EnsureFrontendRequestsAreStateful::class,
            SubstituteBindings::class,
            StartSession::class, // Add this line
        ],

        'web' => [
            // Other middleware
            HandleInertiaRequests::class,
            EncryptCookies::class,
            StartSession::class,
            ShareErrorsFromSession::class,
            SubstituteBindings::class,
            AddQueuedCookiesToResponse::class
        ],
    ];
}
```

- The **Kernel.php** file defines which middleware to apply to API and web routes.
- It ensures smooth handling of sessions, cookies, bindings, and error sharing across the app.
- Middleware like **EnsureFrontendRequestsAreStateful** and **HandleInertiaRequests** are vital for secure and efficient communication between the frontend (Vue.js) and backend (Laravel), especially with stateful requests for authentication.

11. File Storage Structure in storage/app/public/documents



```
storage
  app
    public
      Documents
        Afrikaans
        Business Studies
          Assignments
            Docx to PDF 1.pdf
            PDF 1.pdf
            PDF 5.pdf
          Lecture Notes
            Docx to PDF 1.pdf
            Docx to PDF 2.pdf
            PDF 2.pdf
            PDF 5.pdf
        Computer Science
          Lecture Notes
            Docx to PDF 2.pdf
          Quiz Questions
            Docx to PDF 2.pdf
        Geography
          Assignments
            Docx to PDF 1.pdf
          Lecture Notes
            PDF 2.pdf
        History
          Summaries
            Docx to PDF 1.pdf
        Life Science
```

Purpose:

- This structure follows best practices for file organization, ensuring:
- Easy retrieval and navigation of files based on subject and document type.
- Proper grouping of academic content for different courses.
- Logical and scalable file management, where new subjects or categories can easily be added to the structure.

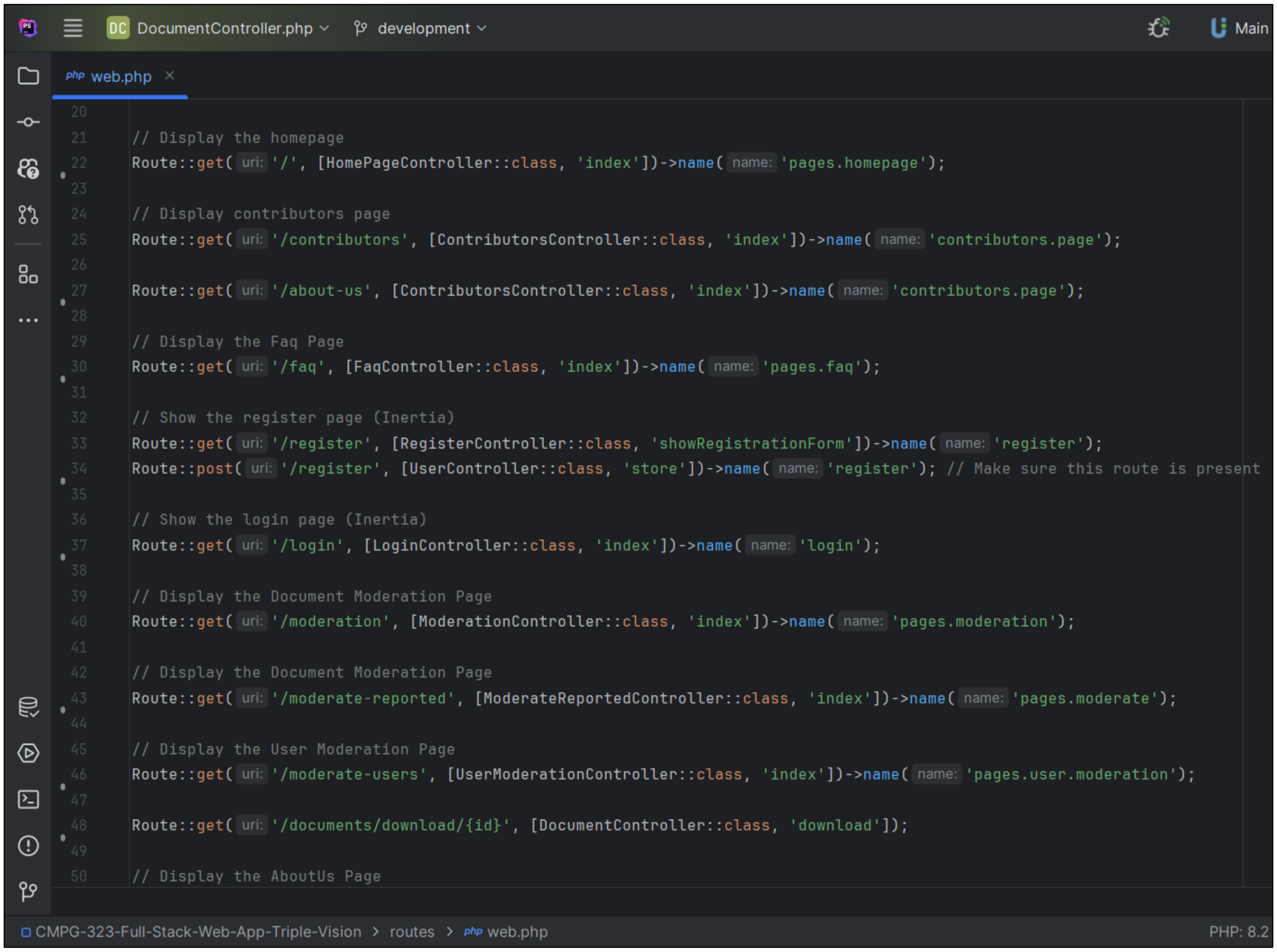
By keeping documents well-organized in the public directory, the Laravel app can efficiently serve and manage user-uploaded files, while still adhering to file access control policies defined in the app.

ENTERING THE FRONTEND:

12. Routes:

We configured the application routing by defining routes in the **routes/api.php** and **routes/web.php** files. This routing setup directs incoming requests to the appropriate controllers and methods, ensuring that the application responds correctly to user interactions and API calls.

Web.php

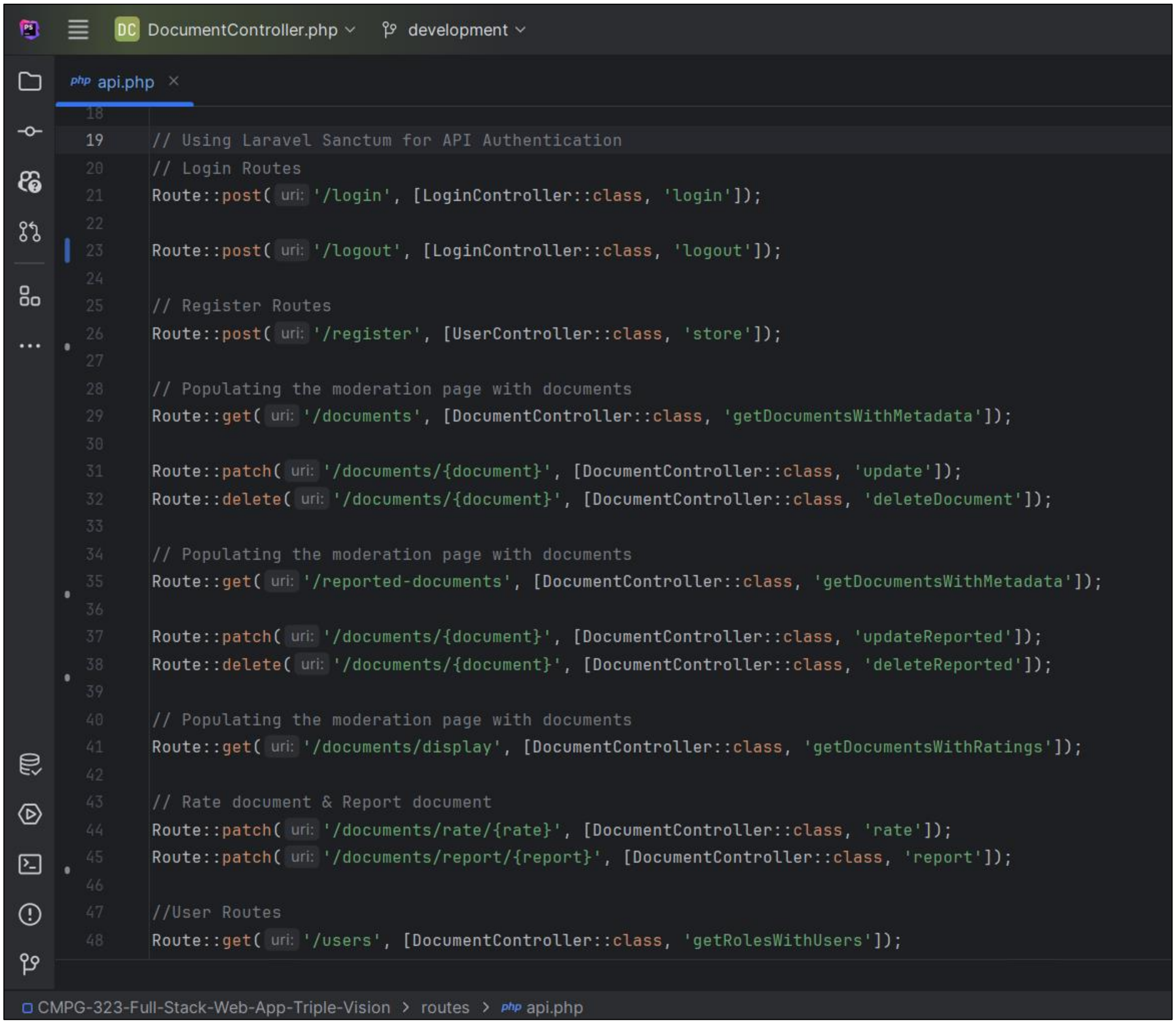


DocumentController.php
development
Main
web.php
CMPG-323-Full-Stack-Web-App-Triple-Vision > routes > web.php
PHP: 8.2

```php

// Display the homepage
Route::get( uri: '/', [HomePageController::class, 'index'])->name( name: 'pages.homepage');

// Display contributors page
Route::get( uri: '/contributors', [ContributorsController::class, 'index'])->name( name: 'contributors.page');

Route::get( uri: '/about-us', [ContributorsController::class, 'index'])->name( name: 'contributors.page');

// Display the Faq Page
Route::get( uri: '/faq', [FaqController::class, 'index'])->name( name: 'pages.faq');

// Show the register page (Inertia)
Route::get( uri: '/register', [RegisterController::class, 'showRegistrationForm'])->name( name: 'register');
Route::post( uri: '/register', [UserController::class, 'store'])->name( name: 'register'); // Make sure this route is present

// Show the login page (Inertia)
Route::get( uri: '/login', [LoginController::class, 'index'])->name( name: 'login');

// Display the Document Moderation Page
Route::get( uri: '/moderation', [ModerationController::class, 'index'])->name( name: 'pages.moderation');

// Display the Document Moderation Page
Route::get( uri: '/moderate-reported', [ModerateReportedController::class, 'index'])->name( name: 'pages.moderate');

// Display the User Moderation Page
Route::get( uri: '/moderate-users', [UserModerationController::class, 'index'])->name( name: 'pages.user.moderation');

Route::get( uri: '/documents/download/{id}', [DocumentController::class, 'download']);

// Display the AboutUs Page
```

The web.php file in a Laravel application defines the routes for handling HTTP requests and directs them to specific controllers. Its primary purpose is to map URLs to their corresponding controller actions, facilitating navigation and functionality within the web app. This includes serving pages, processing form submissions, and managing user interactions, ensuring that the application responds appropriately to user requests.

Api.php

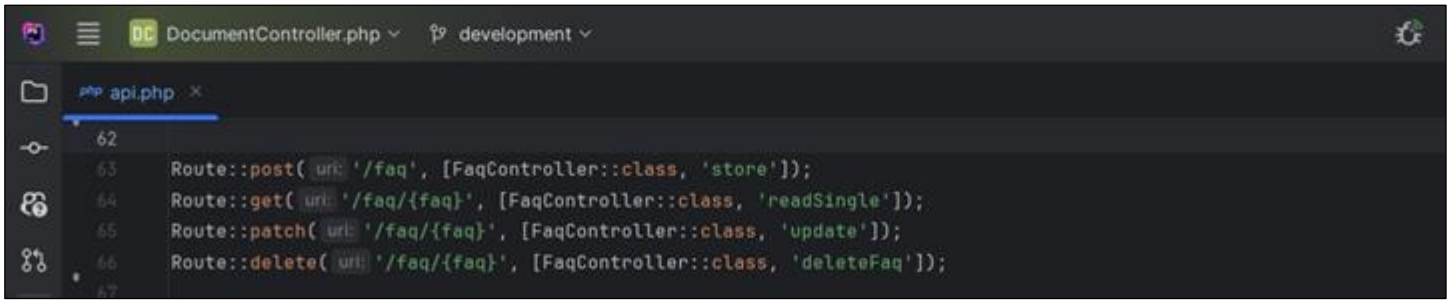

DocumentController.php
development
api.php
CMPG-323-Full-Stack-Web-App-Triple-Vision > routes > api.php

```
// Using Laravel Sanctum for API Authentication
// Login Routes
Route::post( uri: '/login', [LoginController::class, 'login']);

Route::post( uri: '/logout', [LoginController::class, 'logout']);

// Register Routes
Route::post( uri: '/register', [UserController::class, 'store']);

// Populating the moderation page with documents
Route::get( uri: '/documents', [DocumentController::class, 'getDocumentsWithMetadata']);

Route::patch( uri: '/documents/{document}', [DocumentController::class, 'update']);
Route::delete( uri: '/documents/{document}', [DocumentController::class, 'deleteDocument']);

// Populating the moderation page with documents
Route::get( uri: '/reported-documents', [DocumentController::class, 'getDocumentsWithMetadata']);

Route::patch( uri: '/documents/{document}', [DocumentController::class, 'updateReported']);
Route::delete( uri: '/documents/{document}', [DocumentController::class, 'deleteReported']);

// Populating the moderation page with documents
Route::get( uri: '/documents/display', [DocumentController::class, 'getDocumentsWithRatings']);

// Rate document & Report document
Route::patch( uri: '/documents/rate/{rate}', [DocumentController::class, 'rate']);
Route::patch( uri: '/documents/report/{report}', [DocumentController::class, 'report']);

//User Routes
Route::get( uri: '/users', [DocumentController::class, 'getRolesWithUsers']);
```

The Laravel application's web.php file specifies routes for managing common web requests, with an emphasis on page display and form submission management. Its primary functions include managing user-facing routes for navigation and view rendering.

The routes meant for API calls are defined in the api.php file, which is frequently secured for authentication using Laravel Sanctum. These stateless routes generally control data operations **(CRUD),** with an emphasis on moderation features, document management, and user authentication. Instead of utilizing a browser to interact, the client or other systems use the API routes in api.php programmatically.

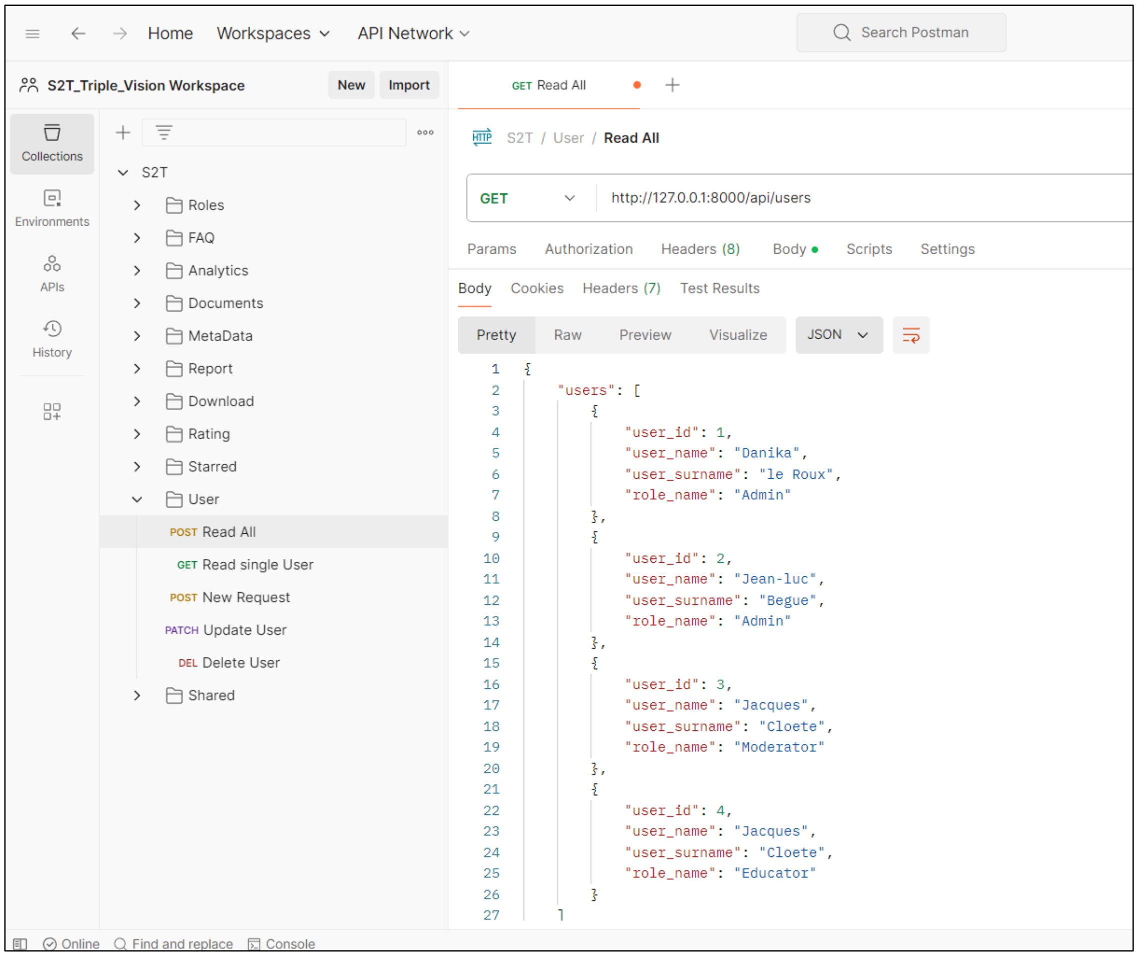
13. The RESTful approach:



```
Route::post( uri: '/faq', [FaqController::class, 'store']);
Route::get( uri: '/faq/{faq}', [FaqController::class, 'readSingle']);
Route::patch( uri: '/faq/{faq}', [FaqController::class, 'update']);
Route::delete( uri: '/faq/{faq}', [FaqController::class, 'deleteFaq']);
```

14. API Testing with Postman:

Postman was utilized for API testing, allowing us to simulate various requests and validate the responses from our application. This tool enables developers to efficiently test endpoints, ensuring that the API behaves as expected and adheres to the defined specifications.



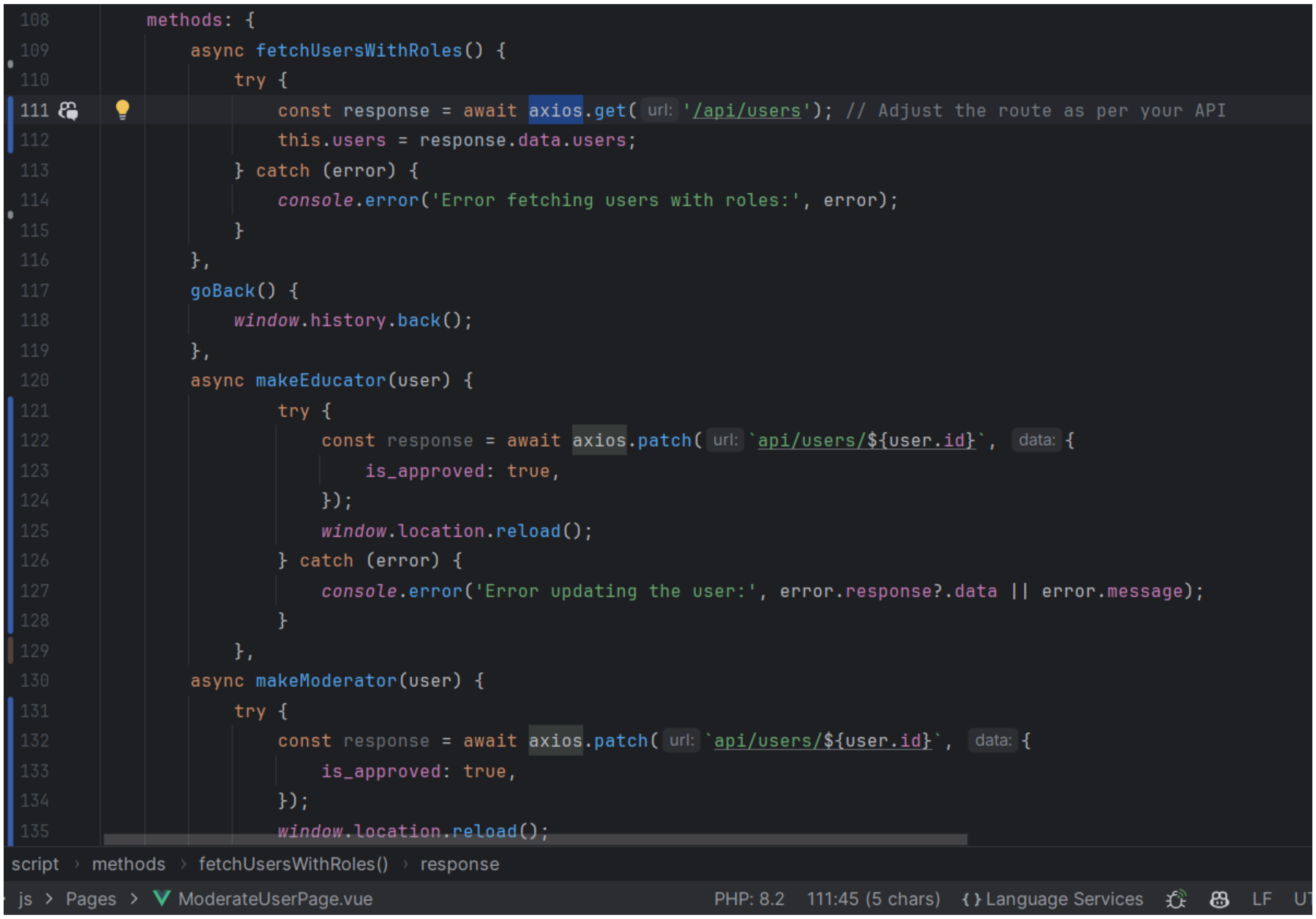
```
GET http://127.0.0.1:8000/api/users
```

```
{
    "users": [
        {
            "user_id": 1,
            "user_name": "Danika",
            "user_surname": "le Roux",
            "role_name": "Admin"
        },
        {
            "user_id": 2,
            "user_name": "Jean-luc",
            "user_surname": "Begue",
            "role_name": "Admin"
        },
        {
            "user_id": 3,
            "user_name": "Jacques",
            "user_surname": "Cloete",
            "role_name": "Moderator"
        },
        {
            "user_id": 4,
            "user_name": "Jacques",
            "user_surname": "Cloete",
            "role_name": "Educator"
        }
    ]
```

15. API Documentation with Editor.Swagger.io:

We documented our API using **Editor.Swagger.io**, creating comprehensive and interactive API documentation. This documentation serves as a reference for developers and stakeholders, outlining the available endpoints, request/response structures, and authentication methods.

16. Using Axios for Asynchronous API Requests in Vue.js



```
    methods: {
        async fetchUsersWithRoles() {
            try {
                const response = await axios.get( url: '/api/users'); // Adjust the route as per your API
                this.users = response.data.users;
            } catch (error) {
                console.error('Error fetching users with roles:', error);
            }
        },
        goBack() {
            window.history.back();
        },
        async makeEducator(user) {
                try {
                    const response = await axios.patch( url: `api/users/${user.id}`,  data: {
                        is_approved: true,
                    });
                    window.location.reload();
                } catch (error) {
                    console.error('Error updating the user:', error.response?.data || error.message);
                }
            },
        async makeModerator(user) {
            try {
                const response = await axios.patch( url: `api/users/${user.id}`,  data: {
                    is_approved: true,
                });
                window.location.reload();
```

script › methods › fetchUsersWithRoles() › response
js › Pages › ModerateUserPage.vue
PHP: 8.2
111:45 (5 chars)
{ } Language Services
LF

Axios is a promise-based HTTP client used in Vue.js to make asynchronous requests to the server. In this example, Axios is used for making GET and PATCH requests to fetch user data and update user roles, respectively. The fetchUsersWithRoles method fetches a list of users from an API, while makeEducator and makeModerator send updates to the server by modifying user roles via PATCH requests. Axios simplifies handling server responses and errors, allowing the app to interact with APIs efficiently, updating data without page reloads.

17. Conclusion:

Comprehensive documentation is crucial for the smooth progression and long-term success of any project. By thoroughly documenting our use of Laravel and Vue.js, database schema, migrations, models, seeders, controllers, routes, and APIs, we have set a solid foundation for efficient development and easy collaboration. This approach not only enhances clarity but also ensures that our code remains maintainable, scalable, and easy to debug as we move closer to the project's completion. Ultimately, well-structured documentation fosters teamwork and streamlines future development efforts.