Document for assignment 15

Task 1: Request Validation

To implement request validation for a registration form with the fields "name," "email," and "password" in Laravel, you can use the validation functionality provided by Laravel. Here's an example of how you can define the validation rules for each field:

```
use Illuminate\Http\Request;

use Illuminate\Support\Facades\Validator;


// ...

public function register(Request $request)

{

  $validator = Validator::make($request->all(), [

    'name' => 'required|string|min:2',

    'email' => 'required|email',

    'password' => 'required|string|min:8',

  ]);


  if ($validator->fails()) {

    return response()->json(['errors' => $validator->errors()], 422);

  }


  // Proceed with registration logic if the validation passes


  // ...

}
```

In the code snippet above, we assume you have a method called register in your controller to handle the registration request. The Request object is injected into the method using Laravel's dependency injection.

Inside the register method, we use the Validator::make() method to create a new validator instance. The first argument is $request->all(), which retrieves all the input data from the request.

The second argument is an array of validation rules. For each field, we specify the required rules using the validation syntax provided by Laravel. For example, 'name' => 'required|string|min:2' means the "name" field is required, must be a string, and must have a minimum length of 2 characters. Similarly, we define the validation rules for the "email" and "password" fields.

After defining the validation rules, we check if the validation fails using $validator->fails(). If the validation fails, we return a JSON response containing the validation errors with a status code of 422 (Unprocessable Entity).

If the validation passes, you can proceed with your registration logic or any other actions you need to perform.

Task 2: Request Redirect

To create a route /home in Laravel that redirects to /dashboard using a 302 redirect, you can use the following code:

```
use Illuminate\Support\Facades\Route;

use Illuminate\Http\RedirectResponse;




// ...


Route::get('/home', function () {
    return new RedirectResponse('/dashboard', 302);
});
```

In the code snippet above, we use the Route::get() method to define the /home route. Inside the route closure, we return a new instance of RedirectResponse class, specifying the target URL /dashboard as the first argument and the HTTP status code 302 (temporary redirect) as the second argument.

This code ensures that when a GET request is made to /home, the browser is redirected to /dashboard with a 302 status code.

Task 3: Global Middleware

To create a global middleware in Laravel that logs the request method and URL for every incoming request to the Laravel log file, you can follow these steps:

Create a new middleware class using the artisan command:

```
php artisan make:middleware LogRequestMiddleware
```

This will generate a new middleware class file called LogRequestMiddleware in the app/Http/Middleware directory.

Open the LogRequestMiddleware class file and implement the logic to log the request information. Here's an example implementation:

```
<?php


namespace App\Http\Middleware;


use Closure;
use Illuminate\Support\Facades\Log;


class LogRequestMiddleware
{
    public function handle($request, Closure $next)
    {
        Log::info('Request:', [
            'method' => $request->method(),
            'url' => $request->fullUrl(),
        ]);


        return $next($request);
    }
}
```

In the handle method, we use the Log facade provided by Laravel to log the request information. We log the request method using $request->method() and the full URL using $request->fullUrl(). You can adjust the logging format or add additional information as needed.

Register the middleware as a global middleware in the App\Http\Kernel class. Open the Kernel.php file located in the app/Http directory and add the middleware to the $middleware property:

```
protected $middleware = [
    // Other middleware...
    \App\Http\Middleware\LogRequestMiddleware::class,
];
```

By adding the LogRequestMiddleware::class to the $middleware property, it becomes a global middleware that will be applied to every incoming request.

Save the changes and you're done! Now, every incoming request will be logged with the request method and URL to the Laravel log file.

Remember to adjust the code according to your specific Laravel application structure and logging preferences.

Task 4: Route Middleware

Open your routes/web.php file or the appropriate routes file in your Laravel application.

Define a route group using the Route::middleware() method and specify the AuthMiddleware as the middleware for the group. Here's an example:

```
use App\Http\Middleware\AuthMiddleware;


// ...


Route::middleware([AuthMiddleware::class])->group(function () {
    Route::get('/profile', function () {
        // Profile route logic here
    });

    Route::get('/settings', function () {
        // Settings route logic here
    });

    // Add more authenticated routes within this group if needed
});
```

In the code snippet above, we import the AuthMiddleware class at the top. Then, we use the Route::middleware() method to define the middleware for the route group, passing AuthMiddleware::class as the argument. This ensures that only authenticated users can access the routes within this group.

Within the route group, we define two routes: /profile and /settings. You can modify the route methods (get, post, etc.) and route logic according to your requirements.

You can also add more authenticated routes within this group if needed by defining additional routes within the group() method

Task 5: Controller

To create a ProductController in Laravel that handles CRUD operations for the Product resource and implements the required methods, you can follow these steps:

Generate the ProductController using the following artisan command:

```
php artisan make:controller ProductController –resource
```

This will generate a new ProductController class with the necessary methods for handling CRUD operations.

Open the ProductController.php file located in the app/Http/Controllers directory.

Modify the ProductController class to implement the required methods. Your ProductController should look like this:

```
namespace App\Http\Controllers;
```

```
use Illuminate\Http\Request;
use App\Models\Product;

class ProductController extends Controller
{
  public function index()
  {
    // Get all products from the database
    $products = Product::all();

    // Return the view with the products data
    return view('products.index', ['products' => $products]);
  }

  public function create()
  {
    // Display the form to create a new product
    return view('products.create');
  }

  public function store(Request $request)
  {
    // Validate the input data from the form
    $validatedData = $request->validate([
      'name' => 'required',
      'price' => 'required|numeric',
      // Add more validation rules for other fields if needed
    ]);

    // Create a new product with the validated data
    $product = Product::create($validatedData);
    // Redirect to the product's details page or any other appropriate route
    return redirect()->route('products.show', ['product' => $product->id]);
}

public function edit($id)
{
    // Find the product by its ID
    $product = Product::findOrFail($id);

    // Display the form to edit an existing product
    return view('products.edit', ['product' => $product]);
}

public function update(Request $request, $id)
{
    // Validate the input data from the form
    $validatedData = $request->validate([
        'name' => 'required',
        'price' => 'required|numeric',
        // Add more validation rules for other fields if needed
    ]);

    // Find the product by its ID
    $product = Product::findOrFail($id);

    // Update the product with the validated data
    $product->update($validatedData);

    // Redirect to the product's details page or any other appropriate route
    return redirect()->route('products.show', ['product' => $product->id]);
    }


    public function destroy($id)

    {

        // Find the product by its ID

        $product = Product::findOrFail($id);


        // Delete the product

        $product->delete();


        // Redirect to the product listing or any other appropriate route

        return redirect()->route('products.index');

    }

}
```

In the code snippet above, we have added the index, create, store, edit, update, and destroy methods to the ProductController. Each method handles a specific CRUD operation for the Product resource.

Note that the implementation assumes you have a Product model and appropriate views (index.blade.php, create.blade.php, edit.blade.php) in the resources/views/products directory. Adjust the views and routes according to your specific Laravel application structure and naming conventions.

Remember to define the routes to map to these controller methods in your routes/web.php file or appropriate routes file.

```
use App\Http\Controllers\ProductController;


Route::resource('products', ProductController::class);
```

Task 6: Single Action Controller

To create a single action controller called ContactController that handles a contact form submission and implements the __invoke() method to process the form submission and send an email with the submitted data, you can follow these steps:

Generate the single action controller using the following artisan command:

```
php artisan make:controller ContactController –invokable
```

This will generate a new ContactController class with the __invoke() method.

Open the ContactController.php file located in the app/Http/Controllers directory.

Modify the ContactController class to implement the __invoke() method. Your ContactController should look like this:

```
namespace App\Http\Controllers;


use Illuminate\Http\Request;

use Illuminate\Support\Facades\Mail;


class ContactController extends Controller

{

  public function __invoke(Request $request)

  {

    // Validate the form data

    $validatedData = $request->validate([

      'name' => 'required',

      'email' => 'required|email',

      'message' => 'required',

    ]);


    // Send an email with the form data

    Mail::to('your-email@example.com')->send(new \App\Mail\ContactForm($validatedData));
```

```
        // Redirect or return a response as needed

        return response()->json(['message' => 'Contact form submitted successfully']);

    }

}
```

In the code snippet above, we have implemented the __invoke() method in the ContactController. This method handles the form submission and sends an email with the submitted data.

The method starts by validating the form data using the $request->validate() method. Adjust the validation rules to match your form fields and requirements.

Next, we use the Mail facade provided by Laravel to send an email. In this example, we send the email to a predefined email address (your-email@example.com) and pass the validated form data to the ContactForm mail class. You'll need to create the ContactForm mail class using the php artisan make:mail ContactForm command and configure it with the appropriate email content.

Finally, you can redirect the user to another page or return a JSON response indicating the success of the form submission.

Remember to adjust the code according to your specific Laravel application structure, email configuration, and email view (ContactForm) implementation.

Note: Don't forget to import the necessary classes at the top of your ContactController file:

```
use Illuminate\Http\Request;

use Illuminate\Support\Facades\Mail;
```

Task 7: Resource Controller

To create a resource controller called PostController that handles CRUD operations for a resource called Post and provides the necessary methods for resourceful routing conventions in Laravel, you can follow these steps:

Generate the PostController using the following artisan command:

```
php artisan make:controller PostController –resource
```

This will generate a new PostController class with the necessary methods for handling CRUD operations.

Open the PostController.php file located in the app/Http/Controllers directory.

Modify the PostController class to implement the required methods. Your PostController should look like this:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Post;

class PostController extends Controller
{
    public function index()
    {
        // Get all posts from the database
        $posts = Post::all();

        // Return the view with the posts data
        return view('posts.index', ['posts' => $posts]);
    }

    public function create()
    {
        // Display the form to create a new post
        return view('posts.create');
    }

    public function store(Request $request)
    {
        // Validate the input data from the form
        $validatedData = $request->validate([
            'title' => 'required',
            'content' => 'required',
            // Add more validation rules for other fields if needed
        ]);

        // Create a new post with the validated data
        $post = Post::create($validatedData);

        // Redirect to the post's details page or any other appropriate route
        return redirect()->route('posts.show', ['post' => $post->id]);
    }

    public function show($id)
    {
        // Find the post by its ID
        $post = Post::findOrFail($id);

        // Return the view with the post data
        return view('posts.show', ['post' => $post]);
    }

    public function edit($id)
    {
        // Find the post by its ID
        $post = Post::findOrFail($id);

        // Display the form to edit an existing post
        return view('posts.edit', ['post' => $post]);
    }

    public function update(Request $request, $id)
    {
        // Validate the input data from the form
        $validatedData = $request->validate([
            'title' => 'required',
            'content' => 'required',
            // Add more validation rules for other fields if needed
        ]);

        // Find the post by its ID
        $post = Post::findOrFail($id);

        // Update the post with the validated data
        $post->update($validatedData);

        // Redirect to the post's details page or any other appropriate route
        return redirect()->route('posts.show', ['post' => $post->id]);
    }

    public function destroy($id)
    {
        // Find the post by its ID
        $post = Post::findOrFail($id);

        // Delete the post
        $post->delete();

        // Redirect to the post listing or any other appropriate route
        return redirect()->route('posts.index');
    }
}
```

In the code snippet above, we have added the index, create, store, show, edit, update, and destroy methods to the PostController. Each method handles a specific CRUD operation for the Post resource.

Note that the implementation assumes you have a Post model and appropriate views (index.blade.php, create.blade.php, edit.blade.php, show.blade.php) in the resources/views/posts directory. Adjust

Task 8: Blade Template Engine

To create a Blade view called welcome.blade.php that includes a navigation bar and a section displaying the text "Welcome to Laravel!", you can follow these steps:

Create a new file named welcome.blade.php in the resources/views directory of your Laravel application

Open the welcome.blade.php file and add the following code:html

```
<!-- resources/views/welcome.blade.php -->


<!DOCTYPE html>

<html>

<head>

  <title>Welcome to Laravel</title>

</head>

<body>

  <!-- Navigation Bar -->

  <nav>

    <!-- Your navigation bar HTML goes here -->

    <ul>

      <li><a href="/">Home</a></li>

      <li><a href="/about">About</a></li>

      <li><a href="/contact">Contact</a></li>

    </ul>

  </nav>
```

```
    <!-- Main Content Section -->

    <section>

        <h1>Welcome to Laravel!</h1>

    </section>

</body>

</html>
```

In the code snippet above, we have created a basic HTML structure for the welcome.blade.php view. Inside the <nav> element, you can add your own HTML code to create the navigation bar, such as an unordered list (<ul>) with list items (<li>) and anchor tags (<a>) for the links.

The main content section is wrapped in a <section> element, and it displays the text "Welcome to Laravel!" using an <h1> heading tag. You can modify the content and styling of this section as needed.

Remember to customize the navigation bar and other parts of the view according to your specific application's requirements and design.

Once you have created the welcome.blade.php view, you can use it in your routes or controllers to display it to the users.

```
use Illuminate\Support\Facades\Route;

Route::get('/', function () {

    return view('welcome');

});
```