Syllabus – Unit 4 -Using Forms and Gathering Input

Creating contact us form

- Create a controller file for the contact page that will contain methods for displaying the contact form view and for processing the data from the contact form
- Create the contact form view
- Add the routes for displaying the form and sending the message
- Create a form request that handles all the validation logic away from your main controllers
- Create a recipient model class that will reference name and email settings in a separate config file to determine where the contact form information is sent
- Create a notification that allows us to format the mail message we will receive
- Add or check the email settings in your .env file

byteXL

Validating user input

- To learn about Laravel's powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user.
- By reading this high-level overview, you'll be able to gain a good general understanding of how to validate incoming request data using Laravel

byteXL

Defining The Routes

- First, let's assume we have the following routes defined in our routes/web.php file:

```php
use App\Http\Controllers\PostController;

Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, 'store']);
```

byteXL

Creating The Controller

- Next, let's take a look at a simple controller that handles incoming requests to these routes. We'll leave the store method empty for now:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Show the form to create a new blog post.
     *
     * @return \Illuminate\View\View
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * Store a new blog post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        // Validate and store the blog post...
    }
}
```

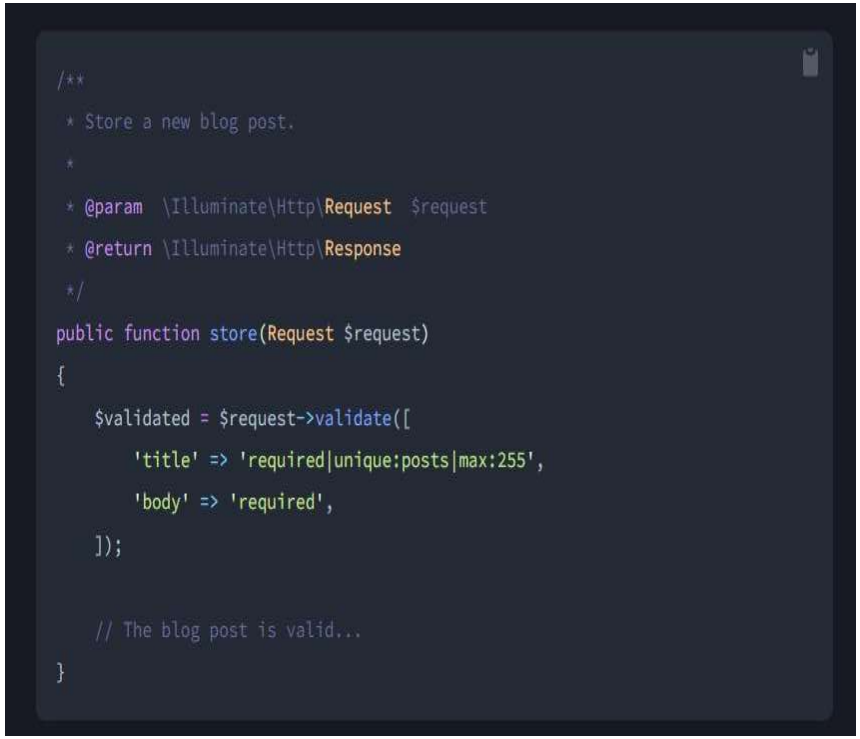
byteXL

Writing The Validation Logic

- Now we are ready to fill in our store method with the logic to validate the new blog post. To do this, we will use the validate method provided by the Illuminate\Http\Request object. If the validation rules pass, your code will keep executing normally; however, if validation fails, an Illuminate\Validation\ValidationException exception will be thrown and the proper error response will automatically be sent back to the user.

- If validation fails during a traditional HTTP request, a redirect response to the previous URL will be generated. If the incoming request is an XHR request, a JSON response containing the validation error messages will be returned.

- To get a better understanding of the validate method, let's jump back into the store method:



```php
/**
 * Store a new blog post.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    $validated = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid...
}
```

byteXL

Displaying The Validation Errors

- So, what if the incoming request fields do not pass the given validation rules? As mentioned previously, Laravel will automatically redirect the user back to their previous location. In addition, all of the validation errors and request input will automatically be flashed to the session.

- An $errors variable is shared with all of your application's views by the Illuminate\View\Middleware\ShareErrorsFromSession middleware, which is provided by the web middleware group. When this middleware is applied an $errors variable will always be available in your views, allowing you to conveniently assume the $errors variable is always defined and can be safely used. The $errors variable will be an instance of Illuminate\Support\MessageBag. For more information on working with this object, check out its documentation.

- So, in our example, the user will be redirected to our controller's create method when validation fails, allowing us to display the error messages in the view:

Displaying The Validation Errors

```blade
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

byteXL

Sending email

- **Laravel uses free feature-rich library SwiftMailer to send emails. Using the library function, we can easily send emails without too many hassles. The e-mail templates are loaded in the same way as views, which means you can use the Blade syntax and inject data into your templates.**

byteXL

Sending email

- **The following table shows the syntax and attributes of send function −**

  Syntax **void send(string|array $view, array $data, Closure|string $callback)**
  Parameters
  $view(string|array) − name of the view that contains email message

  $data(array) − array of data to pass to view

  $callback − a Closure callback which receives a message instance, allowing you to customize the recipients, subject, and other aspects of the mail message

  Returns **nothing**
  Description **Sends email.**

Creating a file uploader, Validating a file uploader,

- . Install Laravel Project
- First, open Terminal and run the following command to create a fresh laravel project:

- composer create-project --prefer-dist laravel/laravel file-upload-laravel
- or, if you have installed the Laravel Installer as a global composer dependency:

- laravel new file-upload-laravel
- 2. Configure Database Details:
- After, Installation Go to the project root directory, open .env file, and set database detail as follow:

byteXL

Creating a file uploader, Validating a file uploader,

- 5. Create Routes
- Go to routes/web.php and create two routes. First, the route handles the form creation, and the second route stores the file in the MySQL database.
- Route::get('/file', [FileController::class, 'index']);
- Route::post('/file', [FileController::class, 'store'])->name('file');
- 6. Create Blade File
- In this step, you need to create a blade view file. Go to resources/views and create one file name fileUpload.blade.php:
- Before starting the application you need to run this command to access all uploaded images ignore this command if you don't upload in a public disk.
- php artisan storage:link
- The public disk is intended for files that are going to be publicly accessible. By default, the public disk uses the local driver and stores these files in storage/app/public. To make them accessible from the web, you should create a symbolic link from public/storage to storage/app/public.

byteXL

Creating a file uploader, Validating a file uploader,

- 3. Create Model and Configure Migration
- php artisan make:model File -m
- Create a Model in laravel, It holds the data definition that interacts with the database.

- 4. Create File Controller
- Now, you need to create a controller name FileController. Use the below command and create a Controller:
- php artisan make:controller FileController
- Next, let's add a method in FileController.php which is located under the app/Http/Controllers folder.
- The first method renders the view via FileUpload controller, and the store() method checks the validation, be it required, mime type, or file size limitation. This method also stores the file into the storage/public/files folder and saves the file name and path in the database.

byteXL

Thank You

byteXL