BUILD A CMS WITH LARAVEL AND VUE - PART 2: IMPLEMENTING POSTS

Neo Ighodaro

October 1st, 2018

> Basic knowledge of Laravel and Vue will be helpful.

In the previous part of this series, we set up user authentication and role authorization but we didn't create any views for the application yet. In this section, we will create the `Post` model and start building the frontend for the application.

Our application allows different levels of accessibility for two kinds of users; the regular user and admin. In this chapter, we will focus on building the view that the regular users are permitted to see.

Before we build any views, let's create the `Post` model as it is imperative to rendering the view.

> The source code for this project is available here on GitHub.

Prerequisites

To follow along with this series, a few things are required:

- Basic knowledge of PHP.
- Basic knowledge of the Laravel framework.

- Basic knowledge of JavaScript (ES6 syntax).
- Basic knowledge of Vue.
- Postman installed on your machine.

Setting up the Post model

We will create the `Post` model with an associated resource controller and a migration file using this command:

```
$ php artisan make:model Post -mr
```

> We added the `r` flag because we want the controller to be a resource controller. The `m` flag will generate a migration for the model.

Let's navigate into the `database/migrations` folder and update the `CreatePostsTable` class that was generated for us:

```
// File: ./app/database/migrations/*_create_posts_table.php
<?php

// [...]

class CreatePostsTable extends Migration
{
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('user_id')->unsigned();
            $table->string('title');
            $table->text('body');
            $table->binary('image')->nullable();
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('posts');
    }
}
```

We included a `user_id` property because we want to create a relationship between the `User` and `Post` models. A `Post` also has an `image` field, which is where its associated image's address will be stored.

Creating a database seeder for the Post table

We will create a new seeder file for the `posts` table using this command:

```
$ php artisan make:seeder PostTableSeeder
```

Let's navigate into the `database/seeds` folder and update the `PostTableSeeder.php` file:

```
// File: ./app/database/seeds/PostsTableSeeder.php
<?php

use App\Post;
use Illuminate\Database\Seeder;

class PostTableSeeder extends Seeder
{
    public function run()
    {
        $post = new Post;
        $post->user_id = 2;
        $post->title = "Using Laravel Seeders";
        $post->body = "Laravel includes a simple method of seeding your da
        $post->save();

        $post = new Post;
        $post->user_id = 2;
        $post->title = "Database: Migrations";
        $post->body = "Migrations are like version control for your databa
        $post->save();
    }
}
```

When we run this seeder, it will create two new posts and assign both of them to the admin user whose ID is 2. We are attaching both posts to the admin user because the regular users are only allowed to view posts and make comments; they can't create a post.

Let's open the `DatabaseSeeder` and update it with the following code:

```
// File: ./app/database/seeds/DatabaseSeeder.php
<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    public function run()
    {
        $this->call([
            RoleTableSeeder::class,
            UserTableSeeder::class,
            PostTableSeeder::class,
        ]);
    }
}
```

> We created the `RoleTableSeeder` and `UserTableSeeder` files in the previous chapter.

We will use this command to migrate our tables and seed the database:

```
$ php artisan migrate:fresh --seed
```

Defining the relationships

Just as we previously created a many-to-many relationship between the `User` and `Role` models, we need to create a different kind of relationship between the `Post` and `User` models.

We will define the relationship as a one-to-many relationship because a user will have many posts but a post will only ever belong to one user.

Open the `User` model and include the method below:

```
// File: ./app/User.php
```

```
    public function posts()
    {
        return $this->hasMany(Post::class);
    }
```

Open the `Post` model and include the method below:

```
    // File: ./app/Post.php
    public function user()
    {
        return $this->belongsTo(User::class);
    }
```

Setting up the routes

At this point in our application, we do not have a front page with all the posts listed. Let's create so anyone can see all of the created posts. Asides from the front page, we also need a single post page in case a user needs to read a specific post.

Let's include two new routes to our `routes/web.php` file:

- The first route will match requests to the root of our application and will be handled by the `PostController@all` action:

```
    Route::get('/', 'PostController@all');
```

> In the `routes/web.php` file, there will already be a route definition for the `/` address, you will have to replace it with the new route definition above.

- The second route will handle requests for specific `Post` items and will be handled by the `PostController@single` action:

```
    Route::get('/posts/{post}', 'PostController@single');
```

With these two new routes added, here's what the `routes/web.php` file should look like this:

```
// File: ./routes/web.php
<?php

Auth::routes();
Route::get('/posts/{post}', 'PostController@single');
Route::get('/home', 'HomeController@index')->name('home');
Route::get('/', 'PostController@all');
```

Setting up the Post controller

In this section, we want to define the handler action methods that we registered in the `routes/web.php` file so that our application know how to render the matching views.

First, let's add the `all()` method:

```
// File: ./app/Http/Controllers/PostController.php
public function all()
{
    return view('landing', [
        'posts' => Post::latest()->paginate(5)
    ]);
}
```

Here, we want to retrieve five created posts per page and send to the `landing` view. We will create this view shortly.

Next, let's add the `single()` method to the controller:

```
// File: ./app/Http/Controllers/PostController.php
public function single(Post $post)
{
    return view('single', compact('post'));
}
```

In the method above, we used a feature of Laravel named route model binding to map the URL parameter to a `Post` instance with the same ID. We are returning a `single` view, which we will create shortly. This will be the view for the single post page.

Building our views

Laravel uses a templating engine called Blade for its frontend. We will use Blade to build these parts of the frontend before switching to Vue in the next chapter.

Navigate to the `resources/views` folder and create two new Blade files:

1. `landing.blade.php`
2. `single.blade.php`

These are the files that will load the views for the landing page and single post page. Before we start writing any code in these files, we want to create a simple layout template that our page views can use as a base.

In the `resources/views/layouts` folder, create a Blade template file and call it `master.blade.php`. This is where we will define the inheritable template for our single and landing pages.

Open the `master.blade.php` file and update it with this code:

```
<!-- File: ./resources/views/layouts/master.blade.php -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, sh
    <meta name="description" content="">
    <meta name="author" content="Neo Ighodaro">
    <title>LaravelCMS</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootst
    <style>
    body {
      padding-top: 54px;
    }
    @media (min-width: 992px) {
      body {
          padding-top: 56px;
      }
    }
    </style>
  </head>
  <body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark fixed-top">
      <div class="container">
        <a class="navbar-brand" href="/">LaravelCMS</a>
```

```
            <div class="collapse navbar-collapse" id="navbarResponsive">
              <ul class="navbar-nav ml-auto">
                 @if (Route::has('login'))
                    @auth
                    <li class="nav-item">
                         <a class="nav-link" href="{{ url('/home') }}">Home</a
                    </li>
                    <li class="nav-item">
                      <a class="nav-link" href="{{ route('logout') }}"
                                          onclick="event.preventDefault();
                                                        document.getElementBy
                        Log out
                      </a>
                      <form id="logout-form" action="{{ route('logout') }}" me
                        @csrf
                      </form>
                     </li>
                     @else
                     <li class="nav-item">
                         <a class="nav-link" href="{{ route('login') }}">Logir
                    </li>
                     <li class="nav-item">
                         <a class="nav-link" href="{{ route('register') }}">Re
                     </li>
                     @endauth
                 @endif
              </ul>
            </div>
          </div>
        </nav>

        <div id="app">
            @yield('content')
        </div>

        <footer class="py-5 bg-dark">
          <div class="container">
            <p class="m-0 text-center text-white">Copyright &copy; LaravelCMS
          </div>
        </footer>
      </body>
    </html>
```

Now we can inherit this template in the `landing.blade.php` file, open it and update it with this code:

```
    {{-- File: ./resources/views/landing.blade.php --}}
```

```
@extends('layouts.master')

@section('content')
<div class="container">
  <div class="row align-items-center">
    <div class="col-md-8 mx-auto">
      <h1 class="my-4 text-center">Welcome to the Blog </h1>

      @foreach ($posts as $post)
      <div class="card mb-4">
        <img class="card-img-top" src=" {!! !empty($post->image) ? '/uploa
        <div class="card-body">
          <h2 class="card-title text-center">{{ $post->title }}</h2>
          <p class="card-text"> {{ str_limit($post->body, $limit = 280, $e
          <a href="/posts/{{ $post->id }}" class="btn btn-primary">Read Mo
        </div>
        <div class="card-footer text-muted">
          Posted {{ $post->created_at->diffForHumans() }} by
          <a href="#">{{ $post->user->name }} </a>
        </div>
      </div>
      @endforeach

    </div>
  </div>
</div>
@endsection
```

Let's do the same with the `single.blade.php` file, open it and update it with this code:

```
{{-- File: ./resources/views/single.blade.php --}}
@extends('layouts.master')

@section('content')
<div class="container">
  <div class="row">
    <div class="col-lg-10 mx-auto">
      <h3 class="mt-4">{{ $post->title }} <span class="lead"> by <a href='
      <hr>
      <p>Posted {{ $post->created_at->diffForHumans() }} </p>
      <hr>
      <img class="img-fluid rounded" src=" {!! !empty($post->image) ? '/up
      <hr>
      <p class="lead">{{ $post->body }}</p>
      <hr>
      <div class="card my-4">
```

```
            <h5 class="card-header">Leave a Comment:</h5>
            <div class="card-body">
              <form>
                <div class="form-group">
                  <textarea class="form-control" rows="3"></textarea>
                </div>
                <button type="submit" class="btn btn-primary">Submit</button>
              </form>
            </div>
          </div>
        </div>
      </div>
    </div>
    @endsection
```

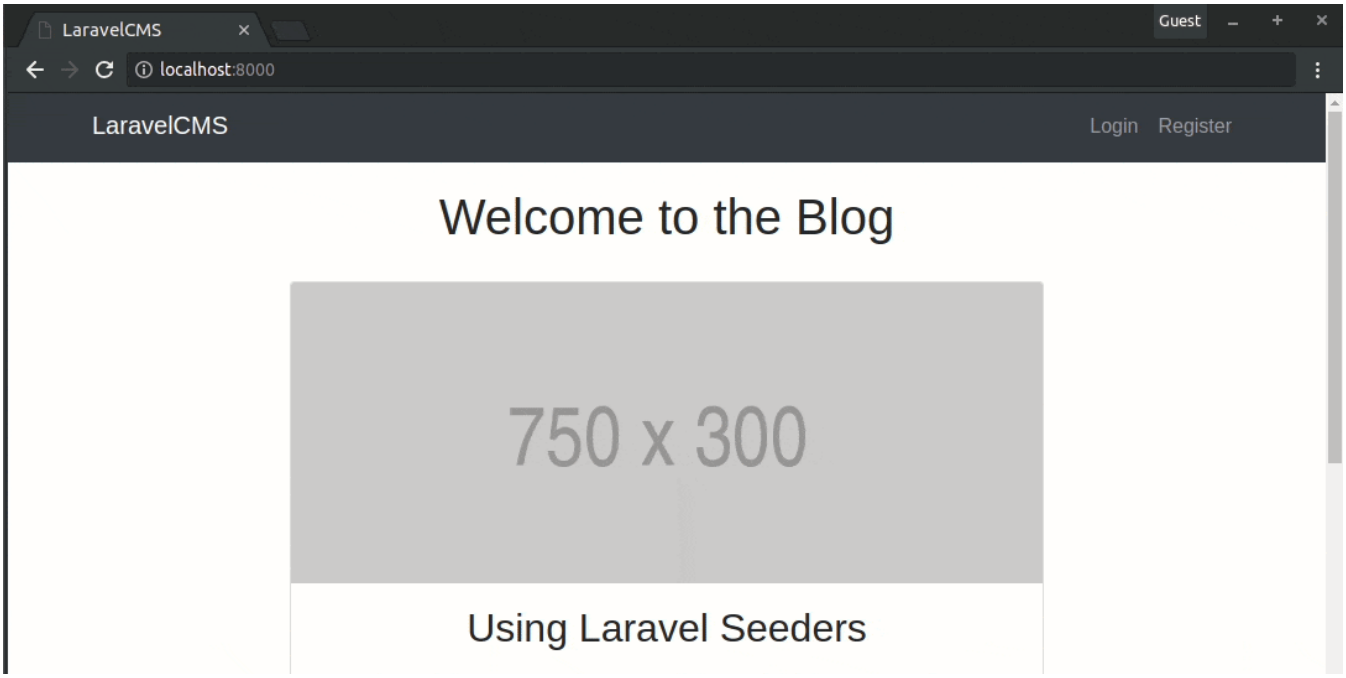
Testing the application

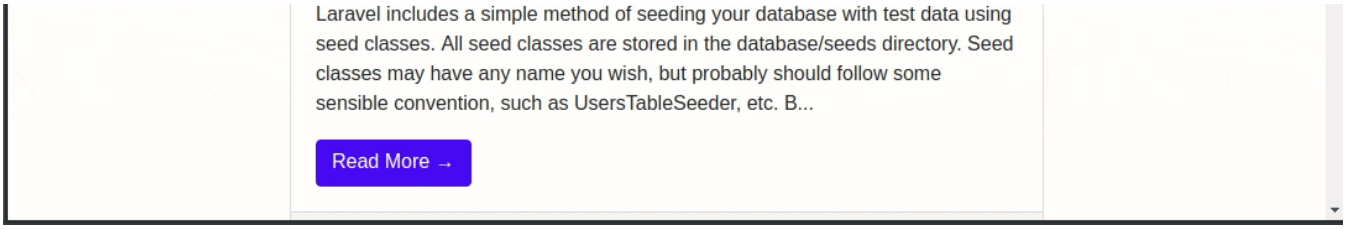
We can test the application to see that things work as we expect. When we serve the application, we expect to see a landing page and a single post page. We also expect to see two posts because that's the number of posts we seeded into the database.

We will serve the application using this command:

```
$ php artisan serve
```

We can visit this address to see the application:





We have used simple placeholder images here because we haven't built the admin dashboard that allows CRUD operations to be performed on posts.

In the coming chapters, we will add the ability for an admin to include a custom image when creating a new post.

Conclusion

In this chapter, we created the `Post` model and defined a relationship on it to the `User` model. We also built the landing page and single page.

In the next part of this series, we will develop the API that will be the medium for communication between the admin user and the post items.

The source code for this project is available here on Github.

JAVASCRIPT LARAVEL PHP VUE.JS

NO PUSHER TECH