WEB 306 – PHP: Databases and Framework

Day 10 – Blade and Forms

1. Blade Templates

1.1 About Blade Templates

Despite the fact that PHP was originally designed specifically to be an HTML templating language, some people do not like the default HTML templating syntax and have created enhanced templating languages for the PHP templating language. Laravel uses their own templating language called Blade which makes creating HTML templates with PHP much easier.

Laravel Resources

1. Laravel Website
2. Laravel Documentation
3. Laracasts
4. Laravel News
5. Scotch.io
6. Stack Overflow

1.2 Base Blade Template

1.2.1 Creating Parent Templates

Unlike with regular PHP templating where we would normally create separate header and footer files, Blade templates allow you to create a “parent” template which can be applied to multiple child templates. We will create a parent template file called `base.blade.php` which we will save in a folder called `layouts` which will be in the `views` folder. This file will start as a normal HTML page.

```
<!DOCTYPE html>
<html lang="en">
  <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width,
          initial-scale=1.0">
```

1.2.2 Using Variables and Functions in Templates

We can echo variables which we have passed to our Blade template using our Controller by simply wrapping them in double curly braces: `{{ $some_variable }}`

```
      <title>{{ $title }}</title>
```

Because of Laravel's folder structure it can not read regular relative links to static files such as CSS files, JavaScript files and image files.

In order to properly link to these files, it is necessary to use the `url()` function where the parameter is the path to the folder which contains the static asset.

```
        <link rel="stylesheet" href="{{ url('css/app.css') }}">
    </head>
```

We can reuse any variables which we passed as many times as we like. The value of the variable will change depending on what value we passed to the matching parameter in our route function on the given page.

```
    <body>
        <header>
            <h1>{{ $title }}</h1>
        </header>
```

Blade automatically does HTML escaping using `htmlentities()` for us so it is not necessary for us to do escaping manually. If we don't want Blade to do HTML escaping then we need to use curly braces with two exclamation points instead:

```
        <footer>
            <p><small>Copyright {!! date('Y') !!}
                <em>{!! $title !!}<em><small></p>
        </footer>
    </body>
</html>
```

1.2.3 Including Partials

We can still simply include template files into our Blade templates using the `@include()` function. Template files which are included using Blade are known as "partials" and should be kept in a separate folder in the `views` folder called `partials`.

Use the directory of the partial file as a parameter of the function. Files in folders can be accessed using a dot between the folder and file name.

```
@include('partials.navigation')
```

1.2.4 Yielding Sections

Blade provides a structure called sections. Sections can be created in parent templates using the `@yeild()` function which is used as a placeholder for content which could be different for every page and then referenced in child tempaltes. Use a name for the section as the parameter of this function.

```
@yield('content')
```

1.3 Blade Child Templates

1.3.1 Extending Parent Templates

Child templates use a parent template as a basis but overwrite any sections which were yielded in the

parent template. We will create a child template called **add-artist.blade.php** which will be saved in the main **resources/views** folder. To specify a parent template to base a child template on, use the **@extends()** function with a string with the path to the parent template as a parameter.

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

1.3.2 Overwriting Content Blocks

In the child template you can use sections with the same names as any of the yield functions from the parent. Any content which is included inside of these sections will overwrite the yield functions in the parent template.

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

@section('content')

@endsection
```

1.3.3 Conditionals

If, if else and if else if statements use a different syntax in Blade templates. They work the same way that they do normally except that they require **@if**, **@else** or **@elseif** keywords and an **@endif** keyword to close the statement. No colons are required. We can check for errors using the **$errors->any()** method and then loop through error messages using the **$errors->all()** method. We can check for and display success messages with session variables using the **session()** function.

```
@if ($errors->any())
<div class="alert alert-danger">
  @foreach ($errors->all() as $error)
  <span class="fa fa-exclamation-triangle"></span> {{ $error }}<br>
  @endforeach
</div>
@elseif( session('success') )
<div class="alert alert-success">
  <span class="fa fa-check"></span> {{ session('success') }}
</div>
@endif
```

1.3.4 Loops

All loops can be used in Blade templates using a slightly different syntax. They work the same way that loops work in PHP except that they require an **@foreach** keywword and an **@endforeach** keyword (if you're using a foreach loop) to close the loop.

1.3.5 Add Form

The Add Artist page will have a form on it with an action attribute which points to the address ***"/store"*** in order to call the **store()** method in our routes file.

All Laravel forms must have a CSRF (Cross-Site-Request-Forgery) token using the **@csrf** statement. A CSRF token is a unique randomized number which is generated each time the form is submitted and is stored both in the submitted form data and in a cookie which is stored in the user's browser.

The CSRF token from the form data is then checked against the CSRF token in the user's browser to ensure that an outside party can not access the data submitted from the form.

```
<form action="{{ url('/store') }}" method="post">
  @csrf

  <label for="name">Name:</label>
  <input type="text" name="name" value="" id="name">

  <select name=”image” id=”image”>
       @foreach ($images as $name => $value)
       <option value=”{{ $value }}”>{{ $name }}</option>
       @endforeach
  </select>

  <label for="styles">Styles:</label>
  <textarea type="text" name="styles" value="" id="styles">
       </textarea>

  <button type="submit"><span class="fa fa-plus"></span>
            Add Artist</button>
</form>
@endsection
```

1.3.6 The View Page

```
@foreach ($artists as $artist)
  <div class="col-3">
       <h2>{{ $artist->name }}</h2>
       <p><img src="{{ url('/img/' . $artist->image) }}"
            alt="{{ $artist->name }}"></p>
       <p>Styles: {{ $artist->styles }}</p>
```

1.3.7 Delete Form

In order for the delete button to work we need to create a form. The form will point to the address ***"/delete"*** in order to call the `delete()` method in our Routes file.

Because HTML forms do not natively accept PUT or DELETE HTTP methods it is necessary to use Laravel's `@method()` function in edit and delete forms in order to "spoof" the functionality of these methods.

```
       <form action="{{ url('/' . $artist->id . '/destroy') }}"
            method="post">
            @method('DELETE')

            @csrf

            <a href="{{ url('/' . $artist->id . '/edit') }}"
                 class="btn btn-dark"><span class="fa fa-edit">
                 </span>Edit</a>
```

```
            <button type="submit"><span class="fa fa-trash"></span>
                Delete</button>
        </form>
    </div>
    @endforeach
</div>
```

1.3.8 Edit Form

The Edit Artist page will have a form on it with an action attribute which points to the address ***"/update"*** in order to call the `update()` method in our Routes file.

Because HTML forms do not natively accept PUT or DELETE HTTP methods it is necessary to use Laravel's `@method()` function in edit and delete forms in order to "spoof" the functionality of these methods.

The value of each input tag will be filled in with the artist's currently defined properties by echoing them into the value attribute.

For the select tag we will loop through and check each option to see if it matches the value of the artist's `$image` property using an if statement. If it does, we will use the `selected` attribute in the tag.

```
<form action="{{ url('/update') }}" method="post">
    @method('PUT')

    @csrf

    <label for="name">Name:</label>
    <input type="text" name="name" value="{{ $artist->name }}"
        id="name">

    <select name=”image” id=”image”>
        @foreach ($images as $name => $value)
        <option value=”{{ $value }}" @if ($name ==
            $artist->image) selected @endif>{{ $name }}</option>
        @endforeach
    </select>

    <label for="styles">Styles:</label>
    <textarea type="text" name="styles" value="{{ $artist->styles }}"
            id="styles"></textarea>

    <button type="submit"><span class="fa fa-edit"></span>
            Edit Artist</button>

</form>
```

2. LaravelCollective

2.1 About LaravelCollective Forms & HTML

Laravel used to include additional PHP classes to be used in Blade templates to generate HTML forms using an alternate syntax. This syntax makes using more complex types of form fields with Laravel easier. However, in version 5 Laravel decided to remove these features from Laravel to make it more lightweight and modular. These classes, and other popular features which were removed from Laravel, are still being maintained as an external library by a group called LaravelCollective.

2.2 Installing LaravelCollective Forms & HTML

To download the Laravel Collective Forms & HTML library use the following Composer command:

```
composer require laravelcollective/html
```

Next, the Laravel Collective Forms & HTML library needs to be listed as a providor. Providors are used to add expanded functionality to Laravel applications. We need to open the **app.php** file which is in the **config** folder and add a new providor to the array of providors.

```
'providers' => [
    // ...providors that come before
    /*
     * Package Service Providers...
     */
    Collective\Html\HtmlServiceProvider::class,
    // ...providors that come after
]
```

In the same file, we also need to add two class "aliases" to the array of aliases.

```
'aliases' => [
    // ...aliases that come before
    /*
     * Package Aliases...
     */
    'Form' => Collective\Html\FormFacade::class,
    'Html' => Collective\Html\HtmlFacade::class,
    // ...aliases that come after
]
```

2.3 Using Laravel Collective Forms

2.3.1 Basic Forms and the Add Form

To create a form using Laravel Collective use the **Form** class alias to access the static **open()** method which is used to create an opening tag for a form. Set **'url' => '/store'** to set the form action. by default, Laravel Collective assumes you are using the **POST** method. Set **'files' => true** for a form which is going to accept a file upload. We are going to convert our image select field into an image upload field so we will include this.

```
{!! Form::open(['url' => '/store', 'files' => true]) !!}
```

Laravel Collective automatically assumes that we want to include a CSRF token in our form so it is not necessary to explicitly include one.

To create individual form field tags, use methods which correspond with the type of input or form field tag such as `label()`, `text()`, `textarea()`, `select()`, `file()`, `hidden()` and `button()`. Each method uses unique parameters to set the attributes of the tag. You can look these up in the Laravel Collective documentation.

```
    <div class="form-group">
        {!! Form::label(['name', 'Name:') !!}
        {!! Form::text(['name', null, ['class' => 'form-control']) !!}

        {!! Form::label(['image', 'Upload Image:') !!}
        {!! Form::file(['image', ['class' => 'btn btn-dark']) !!}
    </div>
    <div class="form-group">
        {!! Form::label(['styles', 'Styles:') !!}
        {!! Form::textarea(['name', null, ['class' => 'form-control',
            'rows' => 5]) !!}
    </div>

    {!! Form::button(['<span class="fa fa-plus"></span> Add Artist',
        ['type' => 'submit', 'class' => 'btn btn-dark']) !!}
{!! Form::close() !!}
```

2.3.2 The Delete Form

For delete forms, you need to include `'method' => 'delete'` to use the `DELETE` method. Delete forms only need a `button()` method but we also want to keep the link to our edit form.

```
@foreach ($artists as $artist)
    <div class="col-3">
        <h2>{{ $artist->name }}</h2>
        <p><img src="{{ url('/img/' . $artist->image) }}"
            alt="{{ $artist->name }}"></p>
        <p>Styles: {{ $artist->styles }}</p>

        {!! Form::open(['url' => '/' . $artist->id . '/destroy',
            'method' => 'delete']) !!}

            <a href="{{ url('/' . $artist->id . '/edit') }}"
                class="btn btn-dark"><span class="fa fa-edit">
                </span>Edit</a>

            {!! Form::button(['<span class="fa fa-trash"></span>
                Delete', ['type' => 'submit', 'class' => 'btn
                btn-dark']) !!}
        {!! Form::close() !!}
    </div>
    @endforeach
```

2.3.3 Model Binding and the Edit Form

For edit forms, you can use the `model()` method of the Form class alias instead of the `open()` method. This sets a model to use as the basis for the form and will fill out all of the form fields with that model's data. It is not necessary to manually fill in the data for most form fields as a result. You do need to include `'method' => 'put'` to use the `PUT` method.

```
{!! Form::model($artist, ['url' => '/' . $artist->id . '/update',
    'method' => 'put', 'files' => true]) !!}
    <div class="form-group">
        {!! Form::label(['name', 'Name:') !!}
        {!! Form::text(['name', null, ['class' => 'form-control']) !!}

        {!! Form::label(['image', 'Upload Image:') !!}
        {!! Form::file(['image', ['class' => 'btn btn-dark']) !!}
```

Because we are accepting images using a `file()` upload input, it is necessary for us to pass the value of the old existing image back to the controller using a `hidden()` field. Laravel Collective can not set this for us automatically because `file()` inputs ignore set value attributes.

```
        {!! Form::hidden(['old_image', $artist->image]) !!}
    </div>
    <div class="form-group">
        {!! Form::label(['styles', 'Styles:') !!}
        {!! Form::textarea(['name', null, ['class' => 'form-control',
            'rows' => 5]) !!}
    </div>

    {!! Form::button(['<span class="fa fa-edit"></span> Edit Artist',
        ['type' => 'submit', 'class' => 'btn btn-dark']) !!}
{!! Form::close() !!}
```

3. Validation and Sanitization Requests

3.1 About Sanitization in Laravel

Laravel provides basic sanitization features such as HTML escaping in Blade templates and a class for trimming strings. However, it leaves most sanitization up to you to write yourself.

3.2 Installing Sanitizer

There are third-party PHP libraries which can be used to simplify sanitization and integrate it into your Laravel app. One of the more popular sanitization libraries is the Sanitizer library.

```
composer require elegantweb/sanitizer
```

Like Laravel Collective Forms & HTML, it is also necessary to register Sanitizer in the array of providers in `config/app.php`.

```
'providers' => [
```

```
    // ...providors that come before
    /*
     * Package Service Providers...
     */
    Collective\Html\HtmlServiceProvider::class,
    Elegant\Sanitizer\Laravel\SanitizerServiceProvider::class,
    // ...providors that come after
  ]
```

We also need to add an alias for the Sanitizer class in the array of aliases which is located in the same file.

```
  'aliases' => [
    // ...aliases that come before
    /*
     * Package Aliases...
     */
    'Form' => Collective\Html\FormFacade::class,
    'Html' => Collective\Html\HtmlFacade::class,
    'Sanitizer' => Elegant\Sanitizer\Laravel\Facade::class,
    // ...aliases that come after
  ]
```

3.3 Requests

3.3.1 Abstract Request

Our actual validation and sanitization would be defined inside of a Laravel Request. To create a Laravel request use the following artisan command.

```
  php artisan make:request Request
```

This request will act as an abstract parent class for other requests and will also use abstract methods.

To review, abstract classes and methods are classes and methods which are not intended to be instantiated, meaning that they are not supposed to be used in an instance of a class to create an object. They are only supposed to be used once they are inherited by a child class.

```
  namespace App\Http\Requests;

  use Illuminate\Foundation\Http\FormRequest;

  abstract class Request extends FormRequest
  {
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    abstract public function authorize();
```

```
    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    abstract public function rules();
}
```

3.3.2 Artist Request

Next we will want to create a request specifically to be used by our `ArtistController`. This request will use the Sanitizer library for sanitization. To create the request use the following command.

```
php artisan make:request ArtistRequest
```

This command will create a new request class with the necessary namespaces and methods already defined.

```
namespace App\Http\Requests;

use App\Http\Requests\Request;
use Elegant\Sanitizer\Laravel\SanitizesInput;

class ArtistRequest extends Request
{
```

3.3.3 Using Traits

The Sanitizer library uses `SanitizesInput` which is a PHP construct known as a trait. Traits are kind of like utility classes which are intended to only be used by other classes. In order to use a trait it is necessary to write the `use` keyword inside of the class which is using it followed by the name of the trait.

```
    use SanitizesInput;
```

3.3.4 Authorization

Laravel automatically generates an `authorize()` method which determines if a user is authorized to make this request. By default it is set to return false. Change this to true.

```
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }
```

3.3.5 Laravel Validation Rules

Laravel also automatically generates a rules method which returns an array of validation rules which must be met in order for a form to submit. These are built-in Laravel rules which are not part of an external library such as Sanitizer.

```
    /**
       * Get the validation rules that apply to the request.
       *
       * @return array
       */
    public function rules()
    {
         return [
              'name' => 'required|regex:([a-zA-Z\-. ]+)|max:255',
              'image' => 'required_without:old_image|nullable|file|image|
                   mimes:jpeg,png,jpg,gif,svg|max:255',
              'styles' => 'required',
          ];
    }
```

Validation rules include:

1. **required - Indicates that a field is required.**
2. **required_without: - Indicates that a filed is required only if another field is blank.**
3. **max - The size limit of a field.**
4. **nullable - Indicates that a form field can be left blank or have a null value - this does not necessarily indicate that the database column itself is also nullable.**
5. **alpha_num - Restricts the data to only aplhanumeric characters**
6. **email - Restricts the data to a string which is formatted as an email address**
7. **regex: - Restricts the data to a string which matches the characters specified in a provided regular expression.**
8. **integer - Restricts the data to only integers**
9. **boolean - Restricts the field to data which can be interpreted as a Boolean. Accepted values are `true`, `false`, `1`, `0`, `"1"` and `"0"`**
10. **file - Restricts the field to a successfully uploaded file.**
11. **image - Restricts the field to an image (jpg, jpeg, gif, bmp or svg file types)**
12. **mimes - Restricts a field to having a MIME type which matches a specified list of file extensions.**

PHP Resources

1. PHP Website
2. XAMPP
3. Documentation
4. W3Schools
5. CSS Tricks
6. Scotch.io
7. Stack Overflow

3.3.6 Sanitizer Filters

Sanitizer automatically generates a method called filters which returns an array of sanitization filters to run data through once it is collected from a form. These are part of WAAVI Sanitizer and are not recognized by Laravel by default.

```
    /**
       * Filters to be applied to the input.
       *
       * @return array
       */
    public function filters()
    {
         return [
              'name' => 'trim|strip_tags|cast:string|capitalize',
              'image' => 'trim|strip_tags',
              'styles' => 'trim|strip_tags|cast:string',
          ];
    }
}
```

Sanitization filters include:

1. **trim - Removes white space before and after a string.**
2. **strip_tags - Strips HTML and PHP tags using `strip_tags()`.**
3. **escape - Runs string through `filter_var()`.**
4. **lowercase - Converts string to all lowercase.**
5. **uppercase - Converts string to all uppercase.**
6. **capitalize - Capitalizes a string.**
7. **cast: - Casts a variable into the given type. Options are: integer, float, string, boolean, object, array and Laravel Collection.**

4. Controller Updates

4.1 Namespaces

Due to the changes we have made to our application, it is necessary for us to make some changes in our `ArtistController` for some of them to take effect. First we need to change the Request namespace to the ArtistRequest namespace instead for our validation and sanitization to work.

```
namespace App\Http\Controllers;

use App\Http\Requests\ArtistRequest;

use App\Artist;

use URL;

class ArtistController extends Controller {
```

4.2 The Store Method

4.2.1 ArtistRequest Class and Validation

We also need to change the name of the `Request` class which our `store()` method uses as a parameter to `ArtistRequest`.

```
  public function store(ArtistRequest $request) {
```

Last week we did validation inside of the controller methods. Since we are doing validation inside of our `ArtistRequest` class now, we no longer need to include validation in the controller.

```
    // $request->validate([
    //         'name' => 'required|max:255',
    //         'image' => 'required|max:255',
    //         'styles' => 'required'
    // ]);

    $artist = new Artist;
    $artist->name = $request->name;
    $artist->styles = $request->styles;
```

4.2.2 Image Upload Processing

Since we are using an image upload instead of a select field, we need to write some code to process the image upload. First we need to check if the submitted form data includes an image file using the `hasFile()` method. Then we can save the file as a variable using the `file()` method.

```
    if ($request->hasFile('image')) {
        $image = $request->file('image');
```

We can format the provided artist name to create an image name by replacing spaces with underscores and changing capital letters to lowercase.

```
        $image_name = strtolower(str_replace(' ', '_',
            $request->name));
```

It is common convention to include a timestamp in file names to prevent files with the same names from conflicting. We can do this using the `time()` function.

```
        $image_time = time();
```

Then we can collect the original file extension of the uploaded image and add it to the end of our new file name.

```
        $image_extension = '.' . $image->getClientOriginalExtension();
        $full_image_name = $image_title . '_' . $image_time .
            $image_extension;
```

We can save the image in a path with folders that correspond to today's date to further keep images organized. This is a convention which the CMS, Wordpress uses. To do this we can use the `date()` function.

```
        $image_path = date('Y/m/d');
```

We need to define a destination path for the file to be moved to using the `public_path()` function. Then we need to move the file using the `move()` method.

```
            $destination_path = public_path('/images/' . $image_path);
            $image->move($destination_path, $full_image_name);
```

Finally, we need to set the `$image` property of the artist to equal the image path concatenated with the full name of the image.

```
            $artist->image = $image_path . '/' . $full_image_name;
        }

        $artist->save();

        return redirect('/')->with(
            'success',
            'New artist, "' . $artist->name . '" added successfully!'
        ]);
    }
```

4.3 The Update Method

4.3.1 ArtistRequest Class and Validation

We also need to change the name of the `Request` class which our `update()` method uses as a parameter to `ArtistRequest`.

```
    public function update(ArtistRequest $request, $id) {
```

Last week we did validation inside of the controller methods. Since we are doing validation inside of our `ArtistRequest` class now, we no longer need to include validation in the controller.

```
        // $request->validate([
        //         'name' => 'required|max:255',
        //         'image' => 'required|max:255',
        //         'styles' => 'required'
        // ]);

        $artist = Artist::find($id);
        $artist->name = $request->name;
        $artist->styles = $request->styles;
```

4.3.2 Image Upload Processing

The code for processing the image upload can stay almost exactly the same.

```
        if ($request->hasFile('image')) {
            $image = $request->file('image');

            $image_name = strtolower(str_replace(' ', '_',
                $request->name));
            $image_time = time();
```

```
            $image_extension = '.' . $image->getClientOriginalExtension();
            $full_image_name = $image_title . '_' . $image_time .
                 $image_extension;

            $image_path = date('Y/m/d');
            $destination_path = public_path('/images/' . $image_path);

            $artist->image = $image_path . '/' . $full_image_name;
```

The one difference is that we need to set the `$image` property of the artist to the value of the hidden `"old_image"` field if no image was uploaded. We can get this as an `$old_image` property of the `$request` object.

```
        } else {
            $artist->image = $request->old_image;
        }

        $artist->save();

        return redirect('/')->with(
            'success',
            'The artist, "' . $artist->name . '" was updated successfully!'
        ]);
    }
```