Views

The essentials

> Of course, it's not practical to return entire HTML documents strings directly from your routes and controllers. Thankfully, views provide a convenient way to place all of our HTML in separate files. (Laravel)

Views help developers separate application logic (php) from the presentation logic (html). In this way, a developer can focus on the presentation structure and styling in one file, while another can focus on the logic behind it. This is the principle of *Separation of Concerns*. Views are files that are stored in the `resources/views` folder. They are written following the *Blade templating language*. Blade is the library that Laravel adopted for this.

How Blade works

A simple view might look like:

```
{{-- View stored in resources/views/greeting.blade.php --}} <html> <body>
<h1>Hello, {{ $name }}</h1> </body> </html>
```

Notice the `{{ $name }}`. This is a piece of Blade syntax that tells Blade to render the content of the variable `$name` in that position. Since this view is stored at `resources/views/greeting.blade.php`, we may return it using the global `view` helper like so:

```
Route::get('/', function () { return view('greeting', ['name' =>
'James']); });
```

Note the `['name' => 'James']`. This lets Blade to make that `$name` variable available when that view is rendered. Now when this route is called, the application will:

1. Call the `view` helper. This helper:
    a. Looks for a file called `greeting.blade.php` in the `resources/views` folder
    b. Makes the `$name` variable available and has `James` assigned to it
    c. Reads the content of the file and replaces `{{ $name }}` with `James`
    d. Returns an HTTP response with the result as content
2. The return value is returned as a HTTP response to the caller

As a result, the returning HTML looks like:

```
<html> <body> <h1>Hello, James</h1> </body> </html>
```

Generating views with Artisan

You may create a view by placing a file with the `.blade.php` extension in your application's `resources/views` directory or by using the `make:view` Artisan command:

```
php artisan make:view greeting
```

Passing data to views

As you saw in the previous examples, you may pass an array of data to views to make that data available to the view:

```
return view('greetings', ['name' => 'Victoria']);
```

When passing information in this manner, the data should be an array with key / value pairs. After providing data to a view, you can then access each value within your view using the data's keys, such as `<?php echo $name; ?>`.

Rendering data

In the previous example, you noticed data being shown using the double curly braces (`{{ ... }}`) statement. This appears to replace the standard PHP `<?php echo ... ?>` syntax, making it a tad easier to read.

But Blade goes beyond this. Rendering data is crucial for applications, yet it can bring security risks, particularly *Cross-Site Scripting (XSS)* attacks. Blade mitigates this risk by using the `{{ ... }}` syntax, which employs PHP's `htmlspecialchars` function, thus averting potential security threats. The following example illustrates this:

```
// When: $name = "<script>alert('Attacked');</script>."; Hello {{ $name }}. // Renders to: Hello &lt;script&gt;alert(&apos;Attacked&apos;);&lt;/script&gt;.
```

This code will replace characters like `<`, `>`, `"`, `'`, and `&` with their corresponding HTML entities (`<`, `>`, `&quot;`, `&apos;`, and `&` respectively). A browser will render these characters on the screen instead of treating them as code.

This works fine in most cases. And you should therefore use this as much as possible. In some cases however, you do need to render data just as is. You need to render data *unescaped*. In this case you need the `{!! ... !!}` statement:

```
// When: $name = "<script>alert('Attacked');</script>."; Hello, {!! $name !!}. // Renders to: Hello <script>alert('Attacked');</script>.
```

This causes the users browser to accept the code as a script and runs it.

> ! Never trust user-provided data blindly. Failure to do so may result in serious security vulnerabilities. Protect your web applications by escaping data properly before displaying it to users.

Comments

When creating views there a lots of times when you want to add comments for structure, clarity and so on. When working with Blade, there are two primary ways of doing this. Eric Barnes created a nice article about this (https://laravel-news.com/laravel-blade-comments), which explains your options here...

Blade comments

The first way is through Blade itself. You can use the following tag pair as a comment:

```
{{-- This comment will not be present in the rendered HTML --}}
```

Under the hood this will compile into something like this:

```
<?php // This comment will not be present in the rendered HTML ?>
```

Because it's a PHP comment the part you comment will not be visible in the source code when someone is viewing it through the browser.

HTML comments

The second way is by using normal HTML comments:

```
<!-- This is a comment -->
```

The difference is this comment will be visible if someone views the source code of you page in the browser.

As a general rule of thumb, you should always go with the Blade style.

Directives

Apart from displaying data, Blade offers handy abbreviations for typical PHP control structures, like conditional statements and loops. These shortcuts - called *directives* - offer a neat, concise method of using PHP control structures while staying familiar to their PHP counterparts. Blade directives can be recognized by the `@` prefix. The two most important must-know directives are explained here:

Conditionals

Conditional directives are used to render specific content if some expression is true, just like normal `if - else` statements. It might look like:

```
@if (count($records) === 1) I have one record! @elseif (count($records) > 1) I have multiple records! @else I don't have any records! @endif
```

Notice that the entire block must be closed with the `@endif` directive, instead of using the `{` `}` curly braces in normal PHP.

Loops

Repetitive directives are used to render specific content a specific number of times:

```
// Renders this 10 times @for ($i = 0; $i < 10; $i++) <p>The current value is {{ $i }}<p> @endfor // Renders this for each user in the collection of users @foreach ($users as $user) <p>This is user {{ $user->id }}</p> @endforeach // Renders this forever @while (true) <p>I'm looping forever.</p> @endwhile
```

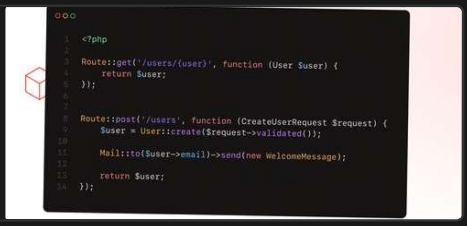
Digging deeper

Most of the above can also be found here:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/views

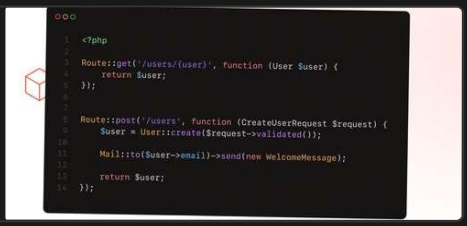


and here:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/blade



Here we highlight some of the topics from these pages that you should know about:

View locations: nested view directories

Views may also be nested within subdirectories of the `resources/views` directory. This is useful for structuring your views. Laravel uses the "Dot" notation to localize these nested views. For example, if your view is stored at `resources/views/admin/profile.blade.php`, you may return it from one of your application's routes / controllers like so:

```
return view('admin.profile');
```

View routes

Sometimes, a route only needs to return a view. For this you should use the `Route::view` method instead of creating a route to a controller method that returns the return value if the `view` helper:

```
Route::view('/welcome', 'greeting');
```

When the user sends a GET request to `/welcome` the app responds with the content of the `greeting.blade.php` view.

If needed, an optional third argument can be used to pass data to the view:

```
Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

Blade Components

Components are special views that contain bits of HTML that can be included anywhere in your views. They provide structure and reusability. You can read about them here:

🔪 Blade components

More Directives

You should be aware of the following directives, because they might come in handy sometimes.

Dealing with empty collections in loops: `@forelse`

A special loop directive can help rendering something specific when the collection to loop is empty:

```
@forelse ($users as $user) <li>{{ $user->name }}</li> @empty <p>No
users</p> @endforelse
```

Conditional classes

Rendering conditional classes is common and might be quite cumbersome and prone to errors, even using the ternary operator. For example:

```
<span class="p-4 {{ $isActive ? 'font-bold' : 'text-gray-500' }} {{
$hasError ? 'bg-red' : '' }}></span>
```

The code actually contains an error. There is a `"` missing, but this is hard to find. Blade has the `@class` directive to help. The same can be done like so:

```
<span @class([ 'p-4', 'font-bold' => $isActive, 'text-gray-500' => !
$isActive, 'bg-red' => $hasError, ])></span>
```

Each class is an entry in an array, and might come with an expression. When so, the class is rendered only if the expression is true.

Conditional attributes

Just like classes, commonly used attributes, that are often rendered conditionally, have their own directives:

For convenience, you may use the `@checked` directive to easily indicate if a given HTML checkbox input is "checked". This directive will echo `checked` if the provided condition evaluates to `true`:

```
<input type="checkbox" name="active" value="active" @checked(old('active',
$user->active)) />
```

Likewise, the `@selected` directive may be used to indicate if a given select option should be "selected":

PHP Copy

```
<select name="version"> @foreach ($product->versions as $version) <option
value="{{ $version }}" @selected(old('version') == $version)> {{ $version
}} </option> @endforeach </select>
```

Further reading

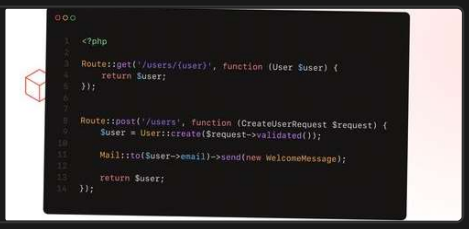
Of course there are many other directives that you may use, like for including subviews and adding raw PHP. Feel free to explore the Laravel documentation for that.

About Views:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/views



About Blade templates:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you