Laravel 5.2 ☰

Blade Templates

Introduction

Blade is the simple, yet powerful templating engine provided with Laravel. Unlike other popular PHP templating engines, Blade does not restrict you from using plain PHP code in your views. All Blade views are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade view files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

Template Inheritance

Defining A Layout

Two of the primary benefits of using Blade are *template inheritance* and *sections*. To get started, let's take a look at a simple example. First, we will examine a "master" page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Blade view:

```
<!-- Stored in resources/views/layouts/master.blade.php -->

<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

As you can see, this file contains typical HTML mark-up. However, take note of the `@section` and `@yield` directives. The `@section` directive, as the name implies, defines a section of content, while the `@yield` directive is used to display the contents of a given section.

Now that we have defined a layout for our application, let's define a child page that inherits the layout.

Extending A Layout

When defining a child page, you may use the Blade `@extends` directive to specify which layout the child page should "inherit". Views which `@extends` a Blade layout may inject content into the layout's sections using `@section` directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using `@yield`:

```
<!-- Stored in resources/views/child.blade.php -->

@extends('layouts.master')

@section('title', 'Page Title')

```

```
@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

In this example, the `sidebar` section is utilizing the `@parent` directive to append (rather than overwriting) content to the layout's sidebar. The `@parent` directive will be replaced by the content of the layout when the view is rendered.

Of course, just like plain PHP views, Blade views may be returned from routes using the global `view` helper function:

```
Route::get('blade', function () {
    return view('child');
});
```

# Displaying Data

You may display data passed to your Blade views by wrapping the variable in "curly" braces. For example, given the following route:

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

You may display the contents of the `name` variable like so:

```
Hello, {{ $name }}.
```

Of course, you are not limited to displaying the contents of the variables passed to the view. You may also echo the results of any PHP function. In fact, you can put any PHP code you wish inside of a Blade echo statement:

```
The current UNIX timestamp is {{ time() }}.
```

> **Note:** Blade `{ }` statements are automatically sent through PHP's `htmlentities` function to prevent XSS attacks.

Blade & JavaScript Frameworks

Since many JavaScript frameworks also use "curly" braces to indicate a given expression should be displayed in the browser, you may use the `@` symbol to inform the Blade rendering engine an expression should remain untouched. For example:

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

In this example, the `@` symbol will be removed by Blade; however, `{{ name }}` expression will remain untouched by the Blade engine, allowing it to instead be rendered by your JavaScript framework.

Echoing Data If It Exists

Sometimes you may wish to echo a variable, but you aren't sure if the variable has been set. We can express this in verbose PHP code like so:

```
{{ isset($name) ? $name : 'Default' }}
```

However, instead of writing a ternary statement, Blade provides you with the following convenient short-cut:

```
{{ $name or 'Default' }}
```

In this example, if the `$name` variable exists, its value will be displayed. However, if it does not exist, the word `Default` will be displayed.

Displaying Unescaped Data

By default, Blade `{{ }}` statements are automatically sent through PHP's `htmlentities` function to prevent XSS attacks. If you do not want your data to be escaped, you may use the following syntax:

```
Hello, {!! $name !!}.
```

Note: Be very careful when echoing content that is supplied by users of your application. Always use the double curly brace syntax to escape any HTML entities in the content.

# Control Structures

In addition to template inheritance and displaying data, Blade also provides convenient short-cuts for common PHP control structures, such as conditional statements and loops. These short-cuts provide a very clean, terse way of working with PHP control structures, while also remaining familiar to their PHP counterparts.

If Statements

You may construct `if` statements using the `@if`, `@elseif`, `@else`, and `@endif` directives. These directives function identically to their PHP counterparts:

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

For convenience, Blade also provides an `@unless` directive:

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

You may also determine if a given layout section has any content using the `@hasSection` directive:

```
<title>
    @hasSection ('title')
        @yield('title') - App Name
    @else
        App Name
    @endif
</title>
```

Loops

In addition to conditional statements, Blade provides simple directives for working with PHP's supported loop structures. Again, each of these directives functions identically to their PHP counterparts:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

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

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

When using loops you might need to end the loop or skip the current iteration:

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

You may also include the condition with the directive declaration in one line:

```
@foreach ($users as $user)
    @continue($user->type == 1)
    <li>{{ $user->name }}</li>

    @break($user->number == 5)
@endforeach
```

Including Sub-Views

Blade's `@include` directive, allows you to easily include a Blade view from within an existing view. All variables that are available to the parent view will be made available to the included view:

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div>
```

Even though the included view will inherit all data available in the parent view, you may also pass an array of extra data to the included view:

```
@include('view.name', ['some' => 'data'])
```

> **Note:** You should avoid using the `__DIR__` and `__FILE__` constants in your Blade views, since they will refer to the location of the cached view.

Rendering Views For Collections

You may combine loops and includes into one line with Blade's `@each` directive:

```
@each('view.name', $jobs, 'job')
```

The first argument is the view partial to render for each element in the array or collection. The second argument is the array or collection you wish to iterate over, while the third argument is the variable name that will be assigned to the current iteration within the view. So, for example, if you are iterating over an array of `jobs`, typically you will want to access each job as a `job` variable within your view partial. The key for the current iteration will be available as the `key` variable within your view partial.

You may also pass a fourth argument to the `@each` directive. This argument determines the view that will be rendered if the given array is empty.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

Comments

Blade also allows you to define comments in your views. However, unlike HTML comments, Blade comments are not included in the HTML returned by your application:

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

# Stacks

Blade also allows you to push to named stacks which can be rendered somewhere else in another view or layout:

```
@push('scripts')
    <script src="/example.js"></script>
@endpush
```

You may push to the same stack as many times as needed. To render a stack, use the `@stack` syntax:

```
<head>
    <!-- Head Contents -->

    @stack('scripts')
</head>
```

# Service Injection

The `@inject` directive may be used to retrieve a service from the Laravel service container. The first argument passed to `@inject` is the name of the variable the service will be placed into, while the second argument is the class / interface name of the service you wish to resolve:

```
@inject('metrics', 'App\Services\MetricsService')

<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

# Extending Blade

Blade even allows you to define your own custom directives. You can use the `directive` method to register a directive. When the Blade compiler encounters the directive, it calls the provided callback with its parameter.

The following example creates a `@datetime($var)` directive which formats a given `$var`:

```
<?php

namespace App\Providers;

use Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function($expression) {
            return "<?php echo with{$expression}->format('m/d/Y H:i'); ?>";
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

As you can see, Laravel's `with` helper function was used in this directive. The `with` helper simply returns the object / value it is given, allowing for convenient method chaining. The final PHP generated by this directive will be:

```
<?php echo with($var)->format('m/d/Y H:i'); ?>
```

After updating the logic of a Blade directive, you will need to delete all of the cached Blade views. The cached Blade views may be removed using the `view:clear` Artisan command.