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. In fact, 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/app.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 view, use the Blade `@extends` directive to specify which layout the child view should "inherit". Views which extend 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.app')

@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.

Contrary to the previous example, this `sidebar` section ends with `@endsection` instead of `@show`. The `@endsection` directive will only define a section while `@show` will define and **immediately yield** the section.

The `@yield` directive also accepts a default value as its second parameter. This value will be rendered if the section being yielded is undefined:

```
@yield('content', View::make('view.name'))
```

Blade views may be returned from routes using the global `view` helper:

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

Components & Slots

Components and slots provide similar benefits to sections and layouts; however, some may find the mental model of components and slots easier to understand. First, let's imagine a reusable "alert" component we would like to reuse throughout our application:

```
<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

The `{{ $slot }}` variable will contain the content we wish to inject into the component. Now, to construct this component, we can use the `@component` Blade directive:

```
@component('alert')
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

To instruct Laravel to load the first view that exists from a given array of possible views for the component, you may use the `componentFirst` directive:

```
@componentFirst(['custom.alert', 'alert'])
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

Sometimes it is helpful to define multiple slots for a component. Let's modify our alert component to allow for the injection of a "title". Named slots may be displayed by "echoing" the variable that matches their name:

```
<!-- /resources/views/alert.blade.php -->
```

```
<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>

    {{ $slot }}
</div>
```

Now, we can inject content into the named slot using the `@slot` directive. Any content not within a `@slot` directive will be passed to the component in the `$slot` variable:

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot

    You are not allowed to access this resource!
@endcomponent
```

Passing Additional Data To Components

Sometimes you may need to pass additional data to a component. For this reason, you can pass an array of data as the second argument to the `@component` directive. All of the data will be made available to the component template as variables:

```
@component('alert', ['foo' => 'bar'])
    ...
@endcomponent
```

Aliasing Components

If your Blade components are stored in a sub-directory, you may wish to alias them for easier access. For example, imagine a Blade component that is stored at `resources/views/components/alert.blade.php`. You may use the `component` method to alias the component from `components.alert` to `alert`. Typically, this should be done in the `boot` method of your `AppServiceProvider`:

```
use Illuminate\Support\Facades\Blade;

Blade::component('components.alert', 'alert');
```

Once the component has been aliased, you may render it using a directive:

```
@alert(['type' => 'danger'])
    You are not allowed to access this resource!
@endalert
```

You may omit the component parameters if it has no additional slots:

```
@alert
    You are not allowed to access this resource!
@endalert
```

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 }}.
```

Blade `{{ }}` statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks.

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() }}.
```

Displaying Unescaped Data

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

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

Be very careful when echoing content that is supplied by users of your application. Always use the escaped, double curly brace syntax to prevent XSS attacks when displaying user supplied data.

Rendering JSON

Sometimes you may pass an array to your view with the intention of rendering it as JSON in order to initialize a JavaScript variable. For example:

```
<script>
    var app = <?php echo json_encode($array); ?>;
</script>
```

However, instead of manually calling `json_encode`, you may use the `@json` Blade directive. The `@json` directive accepts the same arguments as PHP's `json_encode` function:

```
<script>
    var app = @json($array);

    var app = @json($array, JSON_PRETTY_PRINT);
</script>
```

You should only use the `@json` directive to render existing variables as JSON. The Blade templating is based on regular expressions and attempts to pass a complex expression to the directive may cause unexpected failures.

The `@json` directive is also useful for seeding Vue components or `data-*` attributes:

```
<example-component :some-prop='@json($array)'></example-component>
```

Using `@json` in element attributes requires that it be surrounded by single quotes.

HTML Entity Encoding

By default, Blade (and the Laravel `e` helper) will double encode HTML entities. If you would like to disable double encoding, call the `Blade::withoutDoubleEncoding` method from the `boot` method of your `AppServiceProvider`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::withoutDoubleEncoding();
    }
```

```
}
```

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.

The `@verbatim` Directive

If you are displaying JavaScript variables in a large portion of your template, you may wrap the HTML in the `@verbatim` directive so that you do not have to prefix each Blade echo statement with an `@` symbol:

```
@verbatim
    <div class="container">
        Hello, {{ name }}.
    </div>
@endverbatim
```

Control Structures

In addition to template inheritance and displaying data, Blade also provides convenient shortcuts for common PHP control structures, such as conditional statements and loops. These shortcuts 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
```

In addition to the conditional directives already discussed, the `@isset` and `@empty` directives may be used as convenient shortcuts for their respective PHP functions:

```
@isset($records)
    // $records is defined and is not null...
@endisset

@empty($records)
    // $records is "empty"...
@endempty
```

Authentication Directives

The `@auth` and `@guest` directives may be used to quickly determine if the current user is authenticated or is a guest:

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

If needed, you may specify the authentication guard that should be checked when using the `@auth` and `@guest` directives:

```
@auth('admin')
    // The user is authenticated...
@endauth

@guest('admin')
    // The user is not authenticated...
@endguest
```

Section Directives

You may check if a section has content using the `@hasSection` directive:

```
@hasSection('navigation')
    <div class="pull-right">
        @yield('navigation')
    </div>

    <div class="clearfix"></div>
@endif
```

Switch Statements

Switch statements can be constructed using the `@switch`, `@case`, `@break`, `@default` and `@endswitch` directives:

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
@endswitch
```

Loops

In addition to conditional statements, Blade provides simple directives for working with PHP's 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 looping, you may use the loop variable to gain valuable information about the loop, such as whether you are in the first or last iteration through the loop.

When using loops you may also 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
```

The Loop Variable

When looping, a `$loop` variable will be available inside of your loop. This variable provides access to some useful bits of information such as the current loop index and whether this is the first or last iteration through the loop:

```
@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif

    <p>This is user {{ $user->id }}</p>
@endforeach
```

If you are in a nested loop, you may access the parent loop's `$loop` variable via the `parent` property:

```
@foreach ($users as $user)
    @foreach ($user->posts as $post)
        @if ($loop->parent->first)
            This is first iteration of the parent loop.
        @endif
    @endforeach
@endforeach
```

The `$loop` variable also contains a variety of other useful properties:

Property	Description
`$loop->index`	The index of the current loop iteration (starts at 0).
`$loop->iteration`	The current loop iteration (starts at 1).
`$loop->remaining`	The iterations remaining in the loop.
`$loop->count`	The total number of items in the array being iterated.
`$loop->first`	Whether this is the first iteration through the loop.
`$loop->last`	Whether this is the last iteration through the loop.
`$loop->even`	Whether this is an even iteration through the loop.
`$loop->odd`	Whether this is an odd iteration through the loop.
`$loop->depth`	The nesting level of the current loop.
`$loop->parent`	When in a nested loop, the parent's loop variable.

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 --}}
```

PHP

In some situations, it's useful to embed PHP code into your views. You can use the Blade `@php` directive to execute a block of plain PHP within your template:

```
@php
    //
@endphp
```

While Blade provides this feature, using it frequently may be a signal that you have too much logic embedded within your template.

Forms

CSRF Field

Anytime you define an HTML form in your application, you should include a hidden CSRF token field in the form so that the CSRF protection middleware can validate the request. You may use the `@csrf` Blade directive to generate the token field:

```
<form method="POST" action="/profile">
    @csrf

    ...
</form>
```

Method Field

Since HTML forms can't make `PUT`, `PATCH`, or `DELETE` requests, you will need to add a hidden `_method` field to spoof these HTTP verbs. The `@method` Blade directive can create this field for you:

```
<form action="/foo/bar" method="POST">
    @method('PUT')

    ...
</form>
```

Validation Errors

The `@error` directive may be used to quickly check if validation error messages exist for a given attribute. Within an `@error` directive, you may echo the `$message` variable to display the error message:

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

<label for="title">Post Title</label>

<input id="title" type="text" class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

You may pass the name of a specific error bag as the second parameter to the `@error` directive to retrieve validation error messages on pages containing multiple forms:

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email" type="email" class="@error('email', 'login') is-invalid @enderro

@error('email', 'login')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Including Sub-Views

Blade's `@include` directive allows you to include a Blade view from within another 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'])
```

If you attempt to `@include` a view which does not exist, Laravel will throw an error. If you would like to include a view that may or may not be present, you should use the `@includeIf` directive:

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

If you would like to `@include` a view depending on a given boolean condition, you may use the `@includeWhen` directive:

```
@includeWhen($boolean, 'view.name', ['some' => 'data'])
```

To include the first view that exists from a given array of views, you may use the `includeFirst` directive:

```
@includeFirst(['custom.admin', 'admin'], ['some' => 'data'])
```

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

Aliasing Includes

If your Blade includes are stored in a sub-directory, you may wish to alias them for easier access. For example, imagine a Blade include that is stored at `resources/views/includes/input.blade.php` with the following content:

```
<input type="{{ $type ?? 'text' }}">
```

You may use the `include` method to alias the include from `includes.input` to `input`. Typically, this should be done in the `boot` method of your `AppServiceProvider`:

```
use Illuminate\Support\Facades\Blade;

Blade::include('includes.input', 'input');
```

Once the include has been aliased, you may render it using the alias name as the Blade directive:

```
@input(['type' => 'email'])
```

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')
```

Views rendered via `@each` do not inherit the variables from the parent view. If the child view requires these variables, you should use `@foreach` and `@include` instead.

Stacks

Blade allows you to push to named stacks which can be rendered somewhere else in another view or layout. This can be particularly useful for specifying any JavaScript libraries required by your child views:

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

You may push to a stack as many times as needed. To render the complete stack contents, pass the name of the stack to the `@stack` directive:

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

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

If you would like to prepend content onto the beginning of a stack, you should use the `@prepend` directive:

```
@push('scripts')
    This will be second...
@endpush

// Later...

@prepend('scripts')
    This will be first...
@endprepend
```

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 or interface name of the service you wish to resolve:

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

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

Extending Blade

Blade allows you to define your own custom directives using the `directive` method. When the Blade compiler encounters the custom directive, it will call the provided callback with the expression that the directive contains.

The following example creates a `@datetime($var)` directive which formats a given `$var`, which should be an instance of `DateTime`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
```

```
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }
}
```

As you can see, we will chain the `format` method onto whatever expression is passed into the directive. So, in this example, the final PHP generated by this directive will be:

```
<?php echo ($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.

Custom If Statements

Programming a custom directive is sometimes more complex than necessary when defining simple, custom conditional statements. For that reason, Blade provides a `Blade::if` method which allows you to quickly define custom conditional directives using Closures. For example, let's define a custom conditional that checks the current application environment. We may do this in the `boot` method of our `AppServiceProvider`:

```
use Illuminate\Support\Facades\Blade;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Blade::if('env', function ($environment) {
        return app()->environment($environment);
    });
}
```

Once the custom conditional has been defined, we can easily use it on our templates:

```
@env('local')
    // The application is in the local environment...
@elseenv('testing')
    // The application is in the testing environment...
@else
    // The application is not in the local or testing environment...
@endenv
```

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.