IPL
escola superior
de tecnologia e gestão
instituto politécnico
de leiria

Laravel

Laravel Views & Blade

Marco Monteiro

Contributors

- Author(s):
  - Marco Monteiro (marco.monteiro@ipleiria.pt)

Summary

1 – VIEWS

Views

- Views contains HTML - the design of the application
- Views are stored in the "resources/views" folder
- "Dot" notation used to reference nested views.

  For example, view "resources/views/admin/profile.blade.php", is referenced as "**admin.profile**"
- View helper function returns the view:

```
return view('nome', ['var1' => 'value1']);
```

Passing data to the views

- As an **array**
  - **Key** -> variable **name** on the view
  - **Value** -> variable **value** on the view

```
return view('nome', ['var1' => 'value1',
                     'var2' => 'value2']);
```

  - if the variable exists on the controller, we can pass it to the view (maintaining the name) using **compact** function

```
$var1 = 'value1';
$var2 = 'value2';
return view('nome', compact('var1', 'var2'));
```

- *Compact is a php function that creates an array containing variables and their values.*

Passing data to the views

- Using the **with** method
  - With function passes one variable to the view
  - With functions can be **chained** to pass multiple variables

```
return view('nome')->with('var1', 'value1')
                   ->with('var2', 'value2');
```

- We can use a **dynamic name** for the with function.
  It will translate to a variable name in the view

```
return view('nome')->withVar1('value1');
```

withVar1 will create a variable named $var1 in the view

Passing data to the views

- Sharing data with all views
  - Pass data (variables) to all views of your application
  - Using view facade's **share** method
- How?
  - Bootstrap it on AppServiceProvider (*or implement a separate service provider*)
  - On file "app/Providers/AppServiceProvider", add this code:

```
…
use Illuminate\Support\Facades\View;
…
public function boot() {
        View::share('key', 'value');
    }
```

Optimizing Views

- By default, views are compiled on demand
- Since compilation negatively impacts performance, Laravel keeps compiled views on the cache
- To force the compilation of all views of the application:

```
php artisan view:cache
```

- To clear the view cache:

```
php artisan view:clear
```

2 – BLADE

Blade Layout / (Templates)

- Layout (template view) "*inject*" points:
  `@yield``('sectionName')`

- Views use a layout (template), by extending it with
  `@extends``('templateName')`

- View sections will be "*injected*" at one "*inject*" point in the layout (template), with:
  `@section``('templateName')`

Blade Layout / (Templates)

- Layout (template)

```
<!-- Stored in resources/views/layouts/master.blade.php -->
<html>
    <head>
        <title>@yield('title')</title>
    </head>
    <body>
        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

- View

```
@extends('layouts.master')
@section('title', 'Page Title')
@section('content')
    <p>This is my content.</p>
@endsection
```

Blade Comments

- Blade comments

```
{{-- This comment will not be passed on to the HTML --}}
```

- HTML comments

```
<!-- This is a HTML comment.
     It will be passed on to the HTML
     But not displayed on the page -->
```

Blade – Display data

`{{ $a }}` - sanitized (escaped) $a *htmlspecialchars($a)*

- prevents XSS attacks

`{!! $a !!}` - NOT sanitized (unescaped) $a

- DOES NOT prevent XSS attacks

`{{ $a ?? default }}` – using Null Coalescing operator

Blade Directives

- if / elseif / else

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

- unless (*... if not ...*)

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

Blade Directives

- **isset**

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

- **empty**

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

Blade Directives - Loops

- for

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

- foreach

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

- forelse

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

Blade Directives - Loops

- **while**

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

- **continue / break**

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif
    <li>{{ $user->name }}</li>
    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

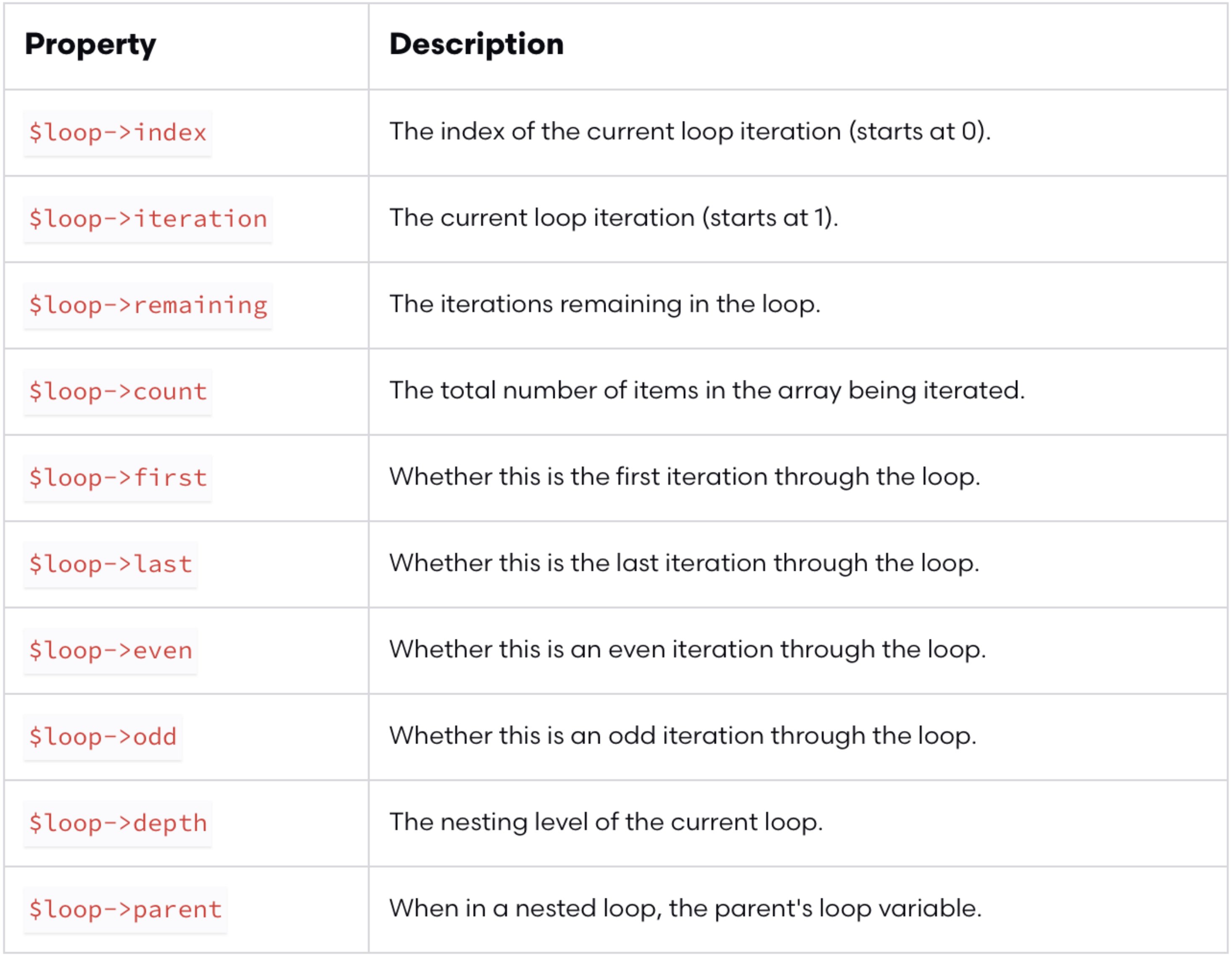
Blade Directives - Loops

- **$loop** variable
  - When looping a $loop variable will be available inside of your 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
```

Blade Directives – Loop Variable



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.

Blade Directives – Debugging

- **dump** - prints variable/s value

```
@dump($varX)
```

```
@dump($var1, $var2)
```

- **dd** (dump & die) – prints variable/s and terminates view processing

```
@dd($varX)
```

```
@dd($var1, $var2)
```

Blade - PHP

- Blade includes a directive to embed PHP code into your views:

```
@php
 $allowEdit = $user->isAdmin() || $user->isTeacher;
@endphp
```

Although Blade provides this feature, using it frequently may be a sign that you have too much logic embedded within your view.

Never use a "normal" PHP block within Blade

```
<?php     . . .
<?=       . . .
```

Blade Directives – class

- **class**

```
@php
    $isActive = false;
    $hasError = true;
@endphp

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

Blade Directives – attributes

- **checked**

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

*Writes "**checked**" attribute when the given expression is true*

- **selected**

```
<select name="fieldName">
    @foreach ($options as $key => $value)
        <option value="{{ $key }}"
            @selected($model->fieldName == $key)>
                {{ $value }}
        </option>
    @endforeach
</select>
```

*Writes "**selected**" attribute when the given expression is true*

Blade Directives – attributes

- **disabled**

```
<button type="submit"
    @disabled($errors->isNotEmpty())>Save </button>
```

*Writes "**disable**" attribute when the given expression is true*

- **readonly**

```
<input … @readonly($user->isNotAdmin()) />
```

*Writes "**readonly**" attribute when the given expression is true*

- **required**

```
<input … @required( $updateMandatory ) />
```

*Writes "**required**" attribute when the given expression is true*

Blade Directives - Forms

- **csrf**
  - Adds a token to prevent CSRF (Cross Site Request Forgeries) attacks
  - By default, it is required for all POST forms

```
<form . . . >
    @csrf
    . . .
</form>
```

Blade Directives - Forms

- **method**
  - HTML forms only support GET or POST method. They can't make a PUT, PATCH or DELETE request
  - @method adds a hidden field (named "_method") to spoof these methods (PUT, PATCH, DELETE):

  Laravel will handle the POST request, as if it was a request with the method specified in the "_method" hidden field

```
<form . . . >
    @method('PUT')
    . . .
</form>
```

PUT

PATCH

DELETE

Blade Directives – Validation Error Messages

- **error('field_name')**
  - The content of @error directive is shown when a validation error message exists (for the given field)

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

- **$message**
  - Within an @error directive, blade creates the variable $message (with the error message)

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

Blade Directives - Authentication

- **auth**

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

```
@auth('admin')
    // The user is authenticated ...
    // and has the authentication guard "admin"
@endauth
```

- **guest**

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

Blade Directives - Authorization

- **can** *(check if user is authorized with a **gate** or a **policy**)*

  More details on "authorization" related content

```
@can('update', $post)
    <!-- The Current User Can Update The Post -->
    <!- 'update' is a gate (or a Post policy) -->
@elsecan('create', App\Post::class)
    <!-- The Current User Can Create New Post -->
    <!- 'create' is a gate (or a Post policy) -->
@endcan
```

- **cannot**

```
@cannot('update', $post)
    <!-- The Current User Can't Update The Post -->
@elsecannot('create', App\Post::class)
    <!-- The Current User Can't Create New Post -->
@endcannot
```

Blade Subviews

- **@include** directive allows you to include a Blade view from within another view

```
@include('shared.errors')
```

  - Included view will inherit all data of parent view
  - It is also possible to add extra data to the included view:

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

- Including a view depending on a given Boolean condition:

when **true:**

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

when **false:**

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

Blade Subviews in Loops

- @each combine loops and includes into one line:

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

Similar to:

```
@foreach($jobs as $job)
   @include('view.name', ['job' => $job])
@endforeach
```

3 – COMPONENTS

Blade View Components

View Components

- **View components** define a section of HTML to include/use in the views

  In that sense, they are similar to a subview, however:

  - They are used with a "special" tag – `<x-alert …/>`
  - They have properties, which are passed on to them as attributes

    ```
    <x-alert type="info" …/>
    ```

  - They include code (each component is associated to a class with the component's **code** and a view with the component's **design**)
  - They support slots. The slot is defined by the component content.

```
<x-alert type="info" …>
   slot content …
</x-alert>
```

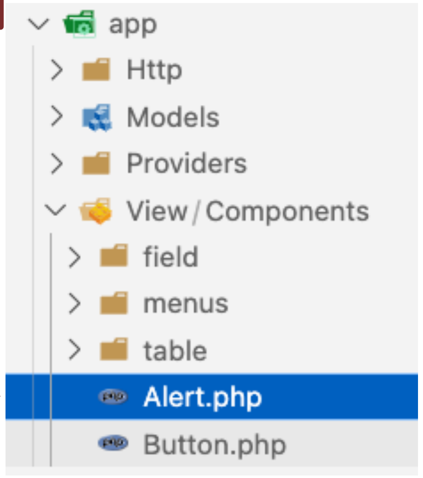
View Components

- Create a View Component with the command:

```
php artisan make:component Alert
```

- Creates the Component's **Class** on this file:

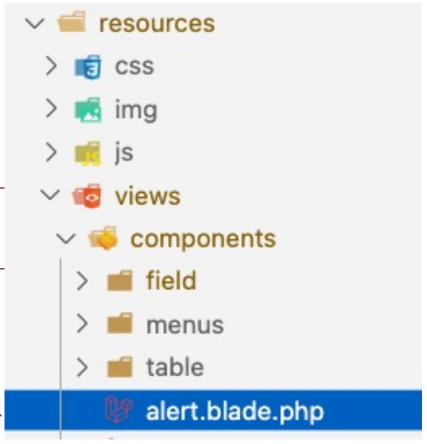
```
app/View/Components/Alert.php
```



app
Http
Models
Providers
View / Components
field
menus
table
Alert.php
Button.php

- Creates the Component's **View** on this file:

```
resources/views/components/alert.blade.php
```



resources
css
img
js
views
components
field
menus
table
alert.blade.php

Component's class

- The component's class defines the **properties** (data), and it might include extra code/methods
  - Properties are defined through the constructor
  - Method render() returns the view (design) of the component

Example

```
class Alert extends Component
{
    public function __construct(
        public string $message,
        public string $type = info,
    ) {}

    public function render(): View
    {
        return view('components.alert');
    }
}
```

Properties:

- message
- type (default = info)

This component uses the view 'components.alert'

Component's view

- The component's view defines the design (HTML)
  - Properties defined on the class can be used on the view

Example:

```
<div class="alert alert-{{ $type }}">
    {{ $message }}
</div>
```

Component usage

- Component can be used by any view or other components
- Component tag start with the prefix **x-**, followed by the kebab case name of the component class:

Examples:

```
<x-alert …/>

<x-user-profile …/>
```

- Component's properties are passed on as attributes:

```
<x-alert type="error" :message="$msg"/>
```

Component's attributes

- Property value can be a hard-coded primitive value (string), using simple HTML attributes strings

```
<x-alert type="error" :message="$msg"/>
```

- Property value can be a dynamic PHP expression (usually a variable), using the prefix : on the attribute

- If the property type is not a string, nor does it convert directly to a string, we have to pass its value using an expression (with the prefix :)
  - Example:

```
<x-select :options="$arrayWithOptions"/>
```

Component's attributes

- We can specify additional attributes to the components (attributes that are not part of the component's constructor)
  - Example:

```
<x-alert type="error" :message="$msg"
         class="mt-4"/>
```

- It is possible to merge attributes specified when using the component, with default values used within the component's view
  - Example (component's view):

```
<div {{ $attributes->merge(['class' => 'p-2 bg-red']) }}>
    {{ $message }}
</div>
```

When using the component with `<x-alert … class="mt-4">`, it will merge the value "mt-4" with the default values within the view ("p-2 bg-red"). The final class value rendered will be: class="**mt-4 p-2 bg-red**"

Component's slot

- We can pass additional content to our component via "**slots**"
- Component slots are rendered by echoing the **$slot** variable within the component's view
  - Example (component usage):

```
<x-alert type="danger">
   <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

We pass content to the slot by injecting **content** into the component

  - Example (component's view):

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

4 – REFERENCES

References

- Official Documentation
  - https://laravel.com/docs/views
  - https://laravel.com/docs/blade

A – BLADE – EXTRA

Extra class:

Just for informational purpose (not required for classes or evaluation)

Blade – Display data

- **Verbatin**
  - To render the {{ … }} into the HTML content. Useful when Javascript frameworks also use the {{ … }} syntax
  - 2 alternatives:

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

```
Hello, @{{$name}}
```

Blade Directives

- hasSection

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

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

Blade Stacks

- Blade allows you to push to named stacks which can be rendered somewhere else in another view or layout.
- Particularly useful for specifying JavaScript libraries required by your child views. On the Layout (Template):

```
<body>
    <!- BODY CONTENT -->
    <script src="common.js"></script>
    @stack('scripts')
</body>
```

- Each views can add an extra script to the stack:

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

Blade Stacks

- Also possible to add the script to the beginning of the stack (using @prepend):

```
@push('scripts')
   This will be second...
   <script src="second.js"></script>
   <script src="third.js"></script>
@endpush

// Later...
@prepend('scripts')
   This will be first...
   <script src="first.js"></script>
@endprepend
```