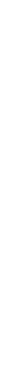
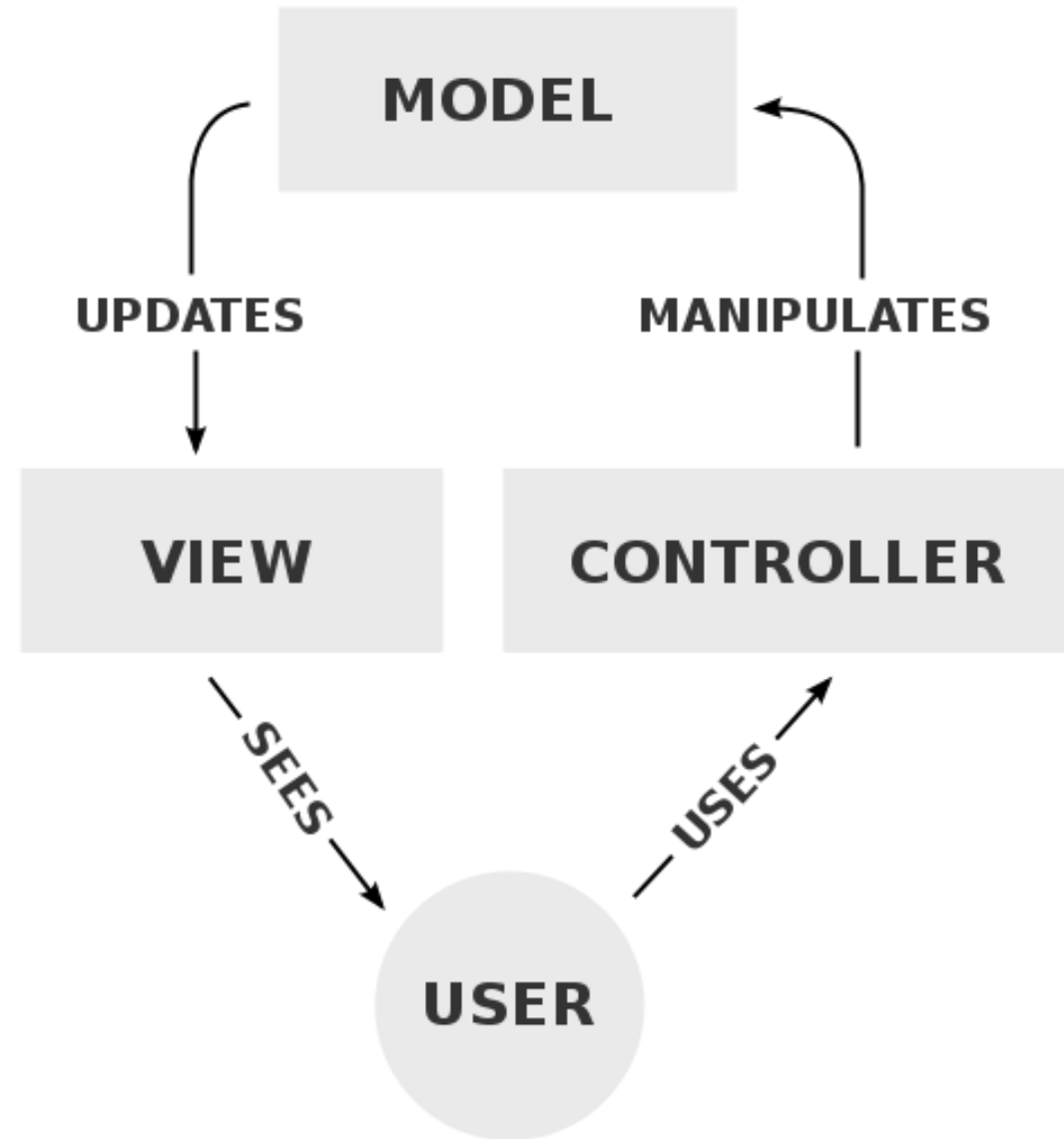




Introduction to Views



Remember
Laravel is an
MVC
Framework



MVC in Laravel

- Models we've covered – these are classes that represent single records in the database
 - ...and since Laravel uses an Active Record design pattern the model also makes changes to the database
- Views we will start looking at today – these are simply webpages which might display a model and/or our UI
- Controllers we will look at later – these group together all the business logic

Routes and Controllers

- When we begin to talk about views it is useful to have a sense of how routes and controllers relate to each other
- The routes\web.php file specifies routes
- In normal practice each route corresponds to a method in a controller
- Your URLs essentially become a set of commands for your controllers



CRUD

- Web applications tend to have items of data in databases
- These items of data (which are represented by our models) are often called resources
- Pretty much every web application will have CRUD actions implemented. Create, Read, Update and Delete.
- **Controllers** then get used to group together the CRUD actions for a particular resource in a single class



Controller Example

- Say you have an Animal model
- You will typically have an Animal controller class – AnimalController
- This will have methods implemented to perform the CRUD actions

Views

Views are our web pages essentially

We could write all the HTML/PHP manually but that would end up creating lots of boilerplate code you'll copy and paste for various projects

Instead we use templating engines and frontend frameworks.

Blade Templates

- Blade is a simple, yet powerful, templating engine which comes with Laravel.
- It is based on/inspired by the Razor C# templating engine which is part of ASP.NET
- Unlike other PHP templating engines you can still use plain PHP code in your views – but perhaps you shouldn't
- All Blade views are compiled into plain PHP code and cached until they are modified – adding close to zero overhead to your application.
- Blade view files use the .blade.php file extension and are typically stored in resources/views

Recap: Passing Data to Views

We can pass data (e.g., from the URL segment) to views:

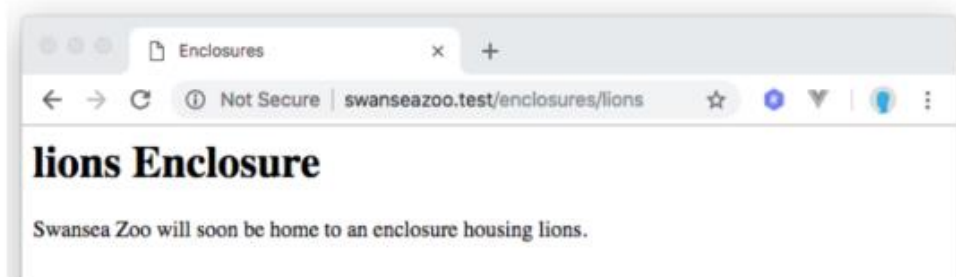
```
Route::get('/enclosures/{animal}', function ($animal) {  
    return view('enclosure', ['animal' => $animal]);  
});
```

We build an associative array and passed it to the view.

```
enclosure.blade.php  
1 <html>  
2 <head>  
3   <title>Enclosures</title>  
4 </head>  
5 <body>  
6   <h1>{{ $animal }} Enclosure</h1>  
7   <p>Swansea Zoo will soon be home to an enclosure housing {{ $animal }}.</p>  
8 </body>  
9 </html>  
10
```

Blade syntax for echoing out a variable within HTML.

The keys of the associative array become the variables in the view.



Views

If we create another view, we could just copy the file and modify it.

This would not be good, we would be **duplicating** lots of HTML. For Example:

- ! Overall Structure
- ! Navigation Links
- ! Header
- ! Footer
- ! Etc

Better to use **Layouts**.

```
enclosure.blade.php
1 <html>
2 <head>
3   <title>Enclosures</title>
4 </head>
5 <body>
6   <h1>{{ $animal }} Enclosure</h1>
7   <p>Swansea Zoo will soon be home to an enclosure hou
8 </body>
9 </html>
10
```

```
enclosure.blade.php
1 <html>
2 <head>
3   <title>Enclosures</title>
4 </head>
5 <body>
6   <h1>{{ $animal }} Enclosure</h1>
7   <p>Swansea Zoo will soon be home to an enc
8 </body>
9 </html>
10
```

Blade Layouts

A primary benefit of using blade is that we can start defining layouts and having views inherit these

This not only helps reduce code but maintains a consistent look across your application.

Layout Files

- Layout files are normally stored in `resources/views/layouts`
- Since we want a consistent look all views should extend these layouts
- Typically we will create one main 'master' layout file which is named
 - `app.blade.php`
 - `resources/views/layouts/app.blade.php`
 - ...this will be used for all pages in our application

Very Basic Layout

- This is a very basic layout.
- [resources/views/layouts/app.blade.php](#)

- It contains the skeleton HTML.
 - For this example we have not used CSS.
- We make holes where content can be injected by the pages.

```
<!doctype html>
<html lang="en">
  <head>
    <title>Swansea Zoo - @yield('title')</title>
  </head>
  <body>
    <h1>Swansea Zoo - @yield('title')</h1>

    <div>
      @yield('content')
    </div>

  </body>
</html>
```

We have another section for the main content area.

We could place the code in the main layout file for navigation, a nice header, footer, etc.

It would then show up on all pages.

@yield directive is used to display the contents of a given section.
In this case a string containing the page title.

```
<!doctype html>
<html lang="en">
  <head>
    <title>Swansea Zoo - @yield('title')</title>
  </head>
  <body>
    <h1>Swansea Zoo - @yield('title')</h1>
    <div>
      @yield('content')
    </div>
  </body>
</html>
```

Using a Layout

This is out `food.blade.php` but now using our layout file.

Specify the layout file using `@extends`.

Dots can be used instead of / in file paths. Make it “look” more **Object Oriented**.

Leave off the `blade.php`.

Specify the content of sections using `@section` and `@endsection`.

Use this form for simple strings.

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

@section('title', 'Food Outlets')

@section('content')
    <p>Swansea Zoo will soon be opening many food outlets – stay tuned!</p>
@endsection
```

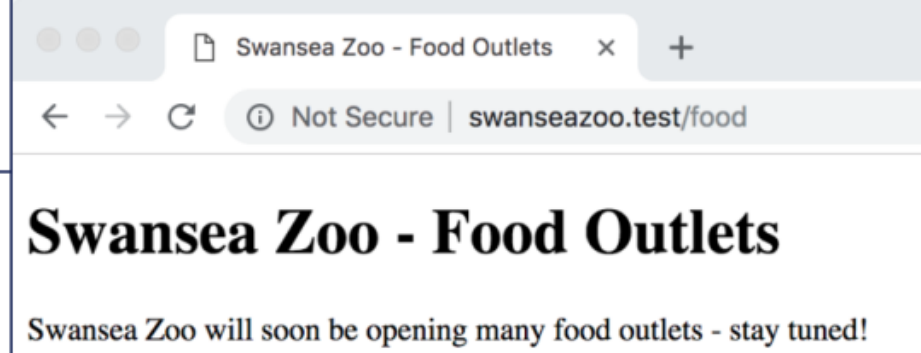
Use this form for structured HTML.

HTML Result

```
<!doctype html>
<html lang="en">
  <head>
    <title>Swansea Zoo - Food Outlets</title>
  </head>
  <body>
    <h1>Swansea Zoo - Food Outlets</h1>

    <div>
      <p>Swansea Zoo will soon be opening many
        food outlets - stay tuned!</p>
    </div>

  </body>
</html>
```



To do this we are going to need: (1) controller methods to get the data, (2) views to show the data, and (3) routes to call the controller methods

Displaying Data

Animal Controller



- As soon as we want to start displaying data it's time to build a controller
- As you might expect we do this with an artisan command
- `php artisan make:controller AnimalController --resource`
- The resource option tells artisan to scaffold methods for a resource
- Controllers end up in the `app/Http/Controllers` folder
- It's up to us to implement the methods in this controller

Routing

- First we can create a route which the user navigates to in order to see all the animals
- As Animal is a resource (the name of the model we care about) then convention is to have a route where the path matches the name of the resource (pluralised)
- For the Animal model the path would be /animals and the full url <http://swanseazoo.test/animals>
- To do this we add the following route...

```
Route::get('animals', 'AnimalController@index');
```

Pattern to match

Controller to use

Method within controller

The AnimalController@index Method

Method should get all the animals and return a web page showing them all

Get all the Animal models.

Pass them to the **animals.index** view.

We organise the views a little. We will place all animal related views in a subfolder named animals.

app/Http/Controllers/AnimalController.php

```
<?php

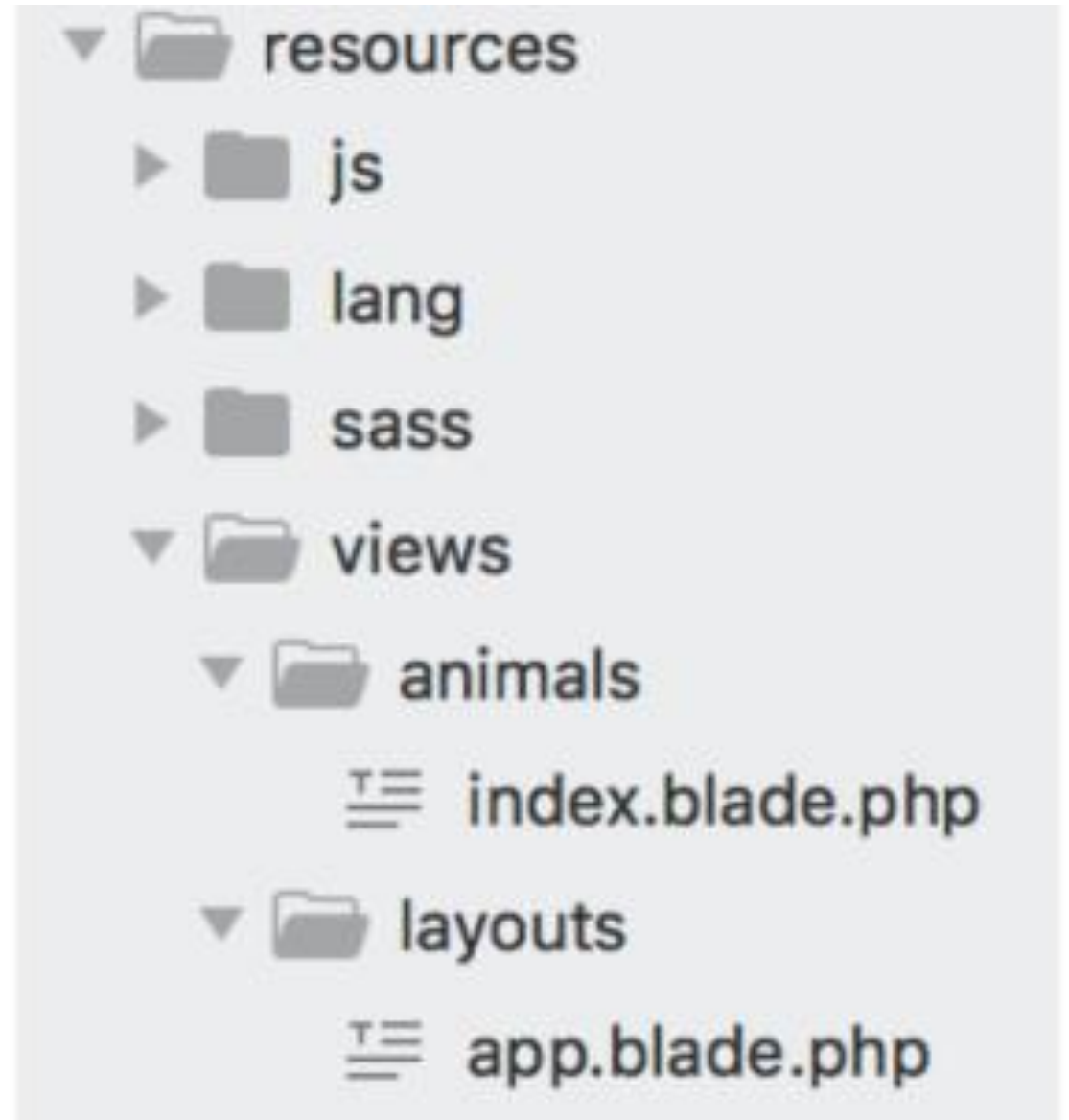
namespace App\Http\Controllers;

use App\Animal;
use Illuminate\Http\Request;

class AnimalController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $animals = Animal::all();
        return view('animals.index', ['animals' => $animals]);
    }
}
```

Creating the view

- The animal.index view is stored in the directory
 - resources\views\animals\index.blade.php
 - When referring to this view in code we only need to write animals.index because we have followed all the conventions



The View animal.index.blade.php (2)

- Views are normally simple.
- Should just contain logic to display data.
- Should do nothing smart!
 - No business logic
 - That goes in controller!

Loop over the array \$animals.
For each iteration the current
animal is stored in the
variable \$animal

Display the current animal's
name as an item in an
unordered list.

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

@section('title', 'Animals')

@section('content')
    <p>The animals of Swansea Zoo:</p>
    <ul>
        @foreach ($animals as $animal)
            <li>{{ $animal->name }}</li>
        @endforeach
    </ul>
@endsection
```

Remember `{{ ... }}` lets you **echo**
out php variables.

Swansea Zoo - Animals

The animals of Swansea Zoo:

- Leo
- Kathryn
- Betty
- Desiree

- When we visit the /animals route a controller method is called
- That method pulls all the records from our database
- It then passes those models to a view
- The view renders those into HTML using the layout files
- All without any manual SQL

```
<!doctype html>
<html lang="en">
  <head>
    <title>Swansea Zoo - Animals</title>
  </head>
  <body>
    <h1>Swansea Zoo - Animals</h1>

    <div>
      <p>The animals of Swansea Zoo:</p>
      <ul>
        <li>Leo</li>
        <li>Kathryn</li>
        <li>Betty</li>
        <li>Desiree</li>
      </ul>
    </div>
  </body>
</html>
```

Displaying Specific Models



Showing details about a specific animal

- The list of animals only includes summary information (the name)
- Now we want to implement the controller action to display full details about a single Animal
- For an Animal resource you would view a specific animal by visiting the following route
 - `/animals/id`
 - Where id is an integer which refers to the id of the resource
 - E.g. to view animal 9 you would use `/animals/9`

Routing

- First we need to set up the route
- In this case we will passing information from the route to the controller method (the id)
- We could name our methods anything but convention is to follow the pattern we have here

```
Route::get('animals/{id}', 'AnimalController@show');
```

Pattern to match

Controller to use

Method within controller

The AnimalController@show Method

We must **show** the model with the given **id**.

Get the animal model with the given **id**.

app/Http/Controllers/AnimalController.php

```
/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    $animal = Animal::findOrFail($id);
    return view('animals.show', ['animal' => $animal]);
}
```

Animal::Find(\$id) would return **null** if no animal exists with the given **id**. Then we would need if statements to deal with that case.

findOrFail is like **find**, but it fails and returns a **404 page** if the model is not found.

Pass the model to the **animals.show** view.

Notice we now use singular variable name.

The View animal.show.blade.php – First Attempt

As a first attempt, we list all the fields of the animal.

It doesn't look fancy, but you can add CSS to jazz it up.

Problem:

- Date of birth was nullable (optional). It looks odd being blank.
- Lets use a **conditional statement** in Blade to make that better

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

@section('title', 'Animal Details')

@section('content')
    <ul>
        <li>Name: {{ $animal->name }}</li>
        <li>Weight: {{ $animal->weight }}kg</li>
        <li>Date of birth: {{ $animal->date_of_birth }}</li>
        <li>Enclosure: {{ $animal->enclosure_id }}</li>
    </ul>
@endsection
```

Swansea Zoo - Animal Details

- Name: Leo
- Weight: 351.6kg
- Date of birth:
- Enclosure: 1

The View animal.show.blade.php – Second Attempt

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

@section('title', 'Animal Details')

@section('content')
    <ul>
        <li>Name: {{ $animal->name }}</li>
        <li>Weight: {{ $animal->weight }}kg</li>

        <li>Date of Birth:
            @if ($animal->date_of_birth)
                {{ $animal->date_of_birth }}
            @else
                Unknown
            @endif
        </li>

        <li>Enclosure: {{ $animal->enclosure_id }}</li>
    </ul>
@endsection
```

Now we use a conditional blade statement to test if the animal's date of birth is not null.

If it exists (not null) we display it.

Otherwise we display a default.

That is a tad verbose for something so simple ...

Swansea Zoo - Animal Details

- Name: Leo
- Weight: 351.6kg
- Date of Birth: Unknown
- Enclosure: 1

The View animal.show.blade.php – Third Attempt

New in PHP 7.

Tests if not null/set, if so return first, otherwise second.

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

@section('title', 'Animal Details')

@section('content')

    <li>Name: {{ $animal->name }}</li>

    <li>Weight: {{ $animal->weight }}</li>

    <li>Date of Birth: {{ $animal->date_of_birth ?? 'Unknown' }}</li>

    <li>Enclosure: {{ $animal->enclosure_id }}</li>

</ul>

@endsection
```

Blade would also allow
{{ \$xyz or 'Default' }}

This checks if \$xyz is set, if so display \$xyz, otherwise display default.

This would **not** work for date of birth as it will be null (and is actually set).

Finally, what about that enclosure

The View animal.show.blade.php – Fourth Attempt

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

@section('title', 'Animal Details')

@section('content')
    <ul>
        <li>Name: {{ $animal->name }}</li>
        <li>Weight: {{ $animal->weight }}kg</li>
        <li>Date of Birth: {{ $animal->date_of_birth ?? 'Unknown' }}</li>
        <li>Enclosure: {{ $animal->enclosure->name }}</li>
    </ul>
@endsection
```

Swansea Zoo - Animal Details

- Name: Leo
- Weight: 351.6kg
- Date of Birth: Unknown
- Enclosure: Central Enclosure

Use no () when you want to execute the relationship

Instead of displaying the id of the closure, we can use the **relationship** to display the enclosures name.

- Later we could even make it a **hyperlink** to the enclosure page.

Question: What if the enclosure is not set?

- It has to be, the migration forced it.

Naming Routes and Linking



Linking to Details

- From the list of animals we should be able to click on a name to visit the details page – in other words make it a hyperlink
 - We could manually construct the anchor tags using model information:

```
@foreach ($animals as $animal)

    <li><a href="/animals/{{ $animal->id }}">{{ $animal->name }}</a></li>


@endforeach
```

Swansea Zoo - Animals

The animals of Swansea Zoo:

- [Leo](#)
- [Kathryn](#)
- [Betty](#)
- [Desiree](#)
- [Herbert](#)

```
<li><a href="http://swanseazoo.test/animals/1">Leo</a></li>
<li><a href="http://swanseazoo.test/animals/2">Kathryn</a></li>
<li><a href="http://swanseazoo.test/animals/3">Betty</a></li>
```



Problems with manual construction

- The method in the last slide was essentially hardcoding the URLs
- What if you change your routing structure? You would have to change it in many places throughout your code.
- Instead Laravel allows us to name routes and use those names when making hyperlinks.

Named Routes

Here we give the route a **name** in the routes file:

```
Route::get('animals/{id}', 'AnimalController@show')  
    ->name('animals.show');
```

(this can be written on one line – line break here for the slides)

- You can use whatever names you like.
- But as your site grows with possibly tens or hundreds of routes, you might need to follow a scheme.
 - I just name them the same as the views.
 - This is the route that shows an animal – simple.

Creating Links with Named Routes

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

@section('title', 'Animals')

@section('content')
    <p>The animals of Swansea Zoo:</p>
    <ul>
        @foreach ($animals as $animal)
            <li><a href="{ route('animals.show', ['id' => $animal->id]) }" >{{ $animal->name }}</a></li>
        @endforeach
    </ul>
@endsection
```

Route function generates URLs.

Route function takes as **first argument** the **name of the route** to generate the URL for.

If route needs parameters, then you pass in **second argument** that is an array of the **route parameters** (variable names and their values).

Here we provide the id of the animal.

Now if the change the URL in the route file it is a single change!

Tasks

1. Starting with the application you have been working on. Pick one of your models, create a controller and views to display a list of all the objects of that type in the database.
2. Now make each item in that list a link to show more details about that particular object – ensure all your views use a constant layout
3. Super hard bonus task – Edit the view showing the list of models so that the list is split into groups over several pages (depending on the total number of items) the user can navigate ... like google search results (see image)

laravel/framework - GitHub

<https://github.com> > [laravel](#) > [framework](#) ▾

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together. ... php framework **laravel**. ... Note: This repository contains the core code of the **Laravel** framework.

Laravel 6 Is Now Released - Laravel News

<https://laravel-news.com> > [laravel-6](#) ▾

6 days ago - The **Laravel** team is proud to announce the release of **Laravel 6** and it's now available to everyone.

Laracasts Journey: Laravel

<https://laracasts.com> > [skills](#) > [laravel](#) ▾

Laravel is a PHP framework for constructing everything from small to enterprise-level applications. As you'll find, it's a joy to use, and just might make you enjoy ...

Searches related to laravel

[laravel download](#)

[laravel documentation](#)

[laravel framework tutorial](#)

[laravel logo](#)

[laravel example](#)

[laravel latest version](#)

[laravel github](#)

[laravel project](#)

