

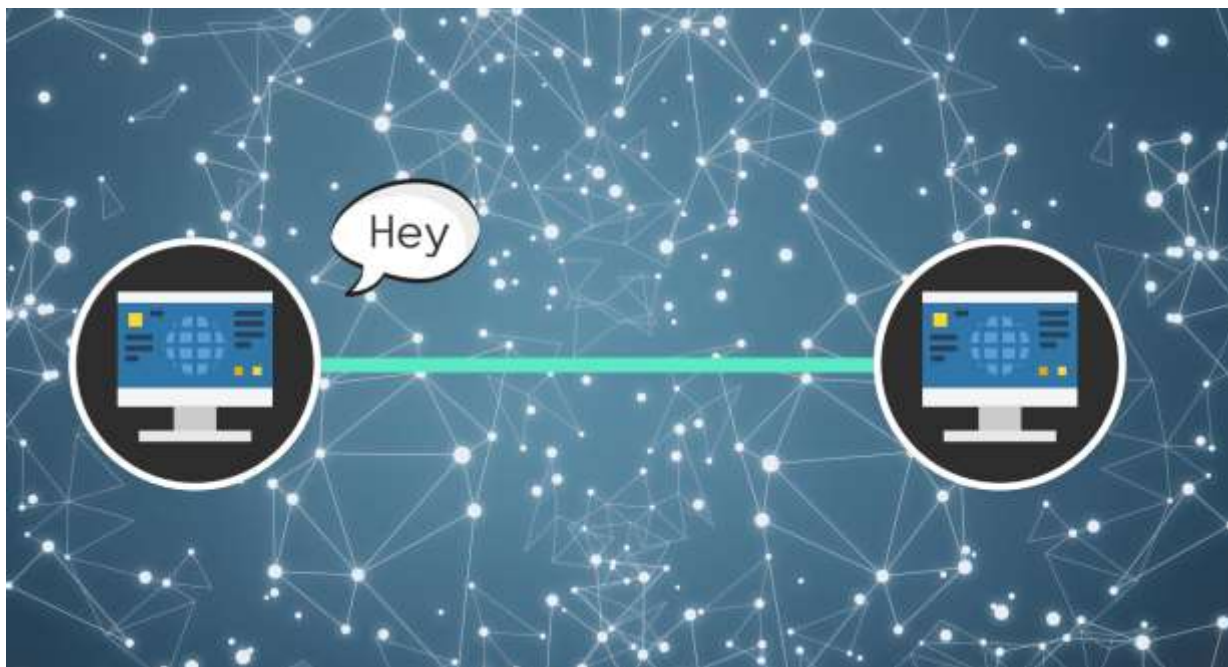


# Software Developer Janeiro\_24

Programação para Web – Server Side

Sara Monteiro

# A internet e o Server Side



Quando mexemos com formulários, dados de tabelas, etc o nosso pc faz uma ligação através do router a um servidor onde são trabalhados e alojados esses dados, e que não são visíveis no browser ao carregar a página.

Esse servidor pode também ser simulado no nosso pc para que possamos testar os nossos projectos sem mexer com o código que está em produção.

# A internet e o Server Side: Pedidos HTTP

- “Hyper Text Transfer Protocol”
- Request -> o Front end faz um pedido ao Backend
- Response -> o Backend envia uma resposta



# Front-End e Back-End

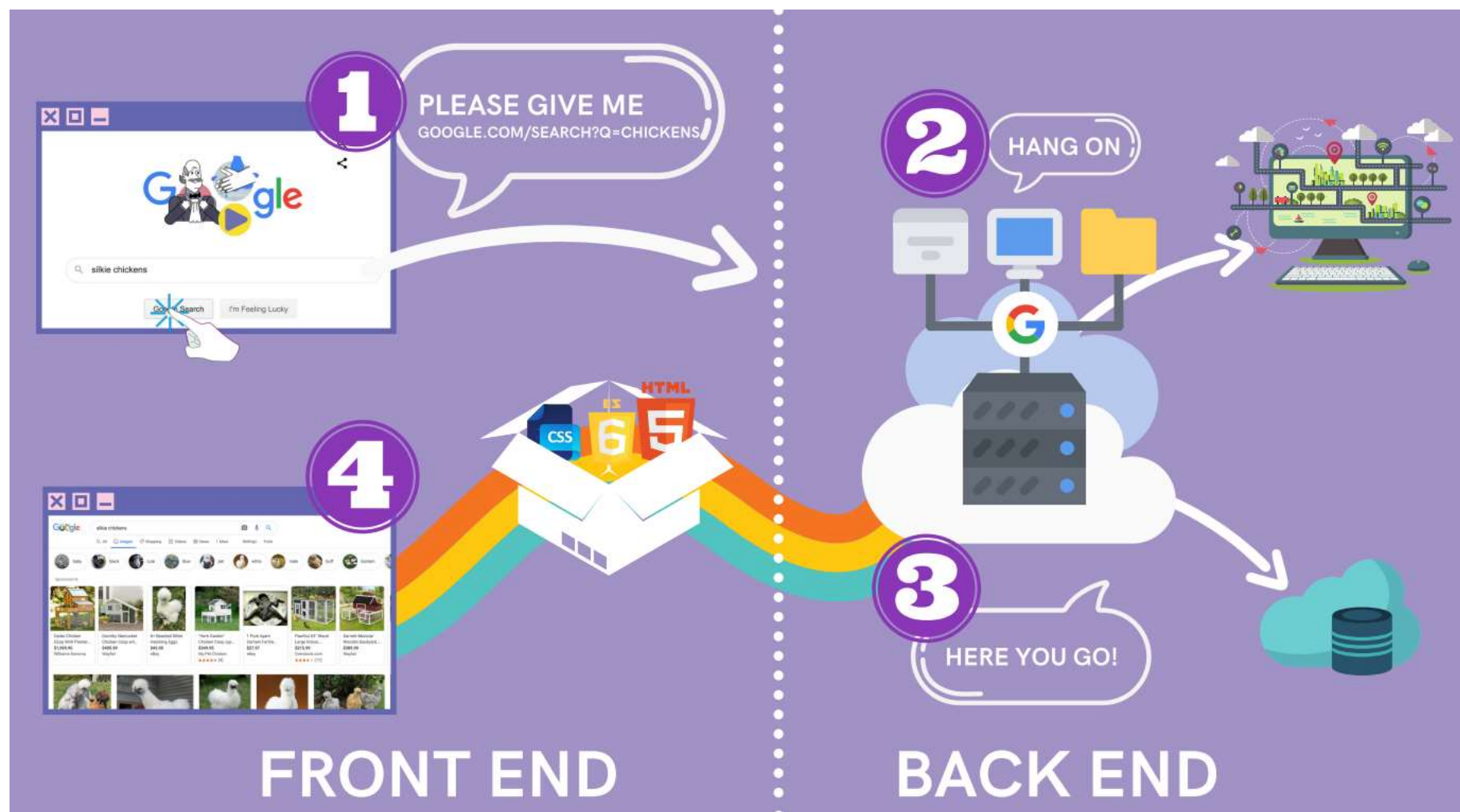


**BACK END**



**FRONT END**

# Front-End e Back-End





# Frameworks e arquitectura MVC



# Frameworks

- conjunto de código / bibliotecas que servem como estrutura para auxiliar no desenvolvimento de software
- funcionalidades já determinadas para agilizar o processo e evitar que os programadores as tenham que reescrever (ex: login, autenticação, etc)
- as Frameworks permitem desenvolver código Full Stack e as MicroFrameworks são pequenos módulos de simplificação, como o [Lumen](#)



# Laravel, o que é

- Uma framework baseada no modelo MVC PHP.
- Criada por Taylor Otwell.
- Conjunto de funcionalidades pré criadas a que podemos aceder de modo a construir uma aplicação com Server Side de uma forma mais rápida.

[Documentação](#)





# Porquê Laravel

- Linguagem core é PHP, linguagem simples e de boa integração na Web
- PHP usado em CMS como Wordpress
- Uma framework baseada no modelo MVC que é usada na maior parte das estruturas de Server Side como Golang, Node, .Net, etc..
- Sistemas de Rotas como Endpoints
- Sistema de migrações usado na maior parte das estruturas de ligação de dados
- Fácil uso para principiantes, mas facilmente escalável para níveis mais aprofundados de desenvolvimento

# A arquitectura MVC

MVC (Model-View-Controller) é uma forma de estruturar o nosso software e que nos permite dividir a aplicação em três camadas.

- **Model** : modelagem de dados e regras de negócio. É nela que constam as classes, consultas à BD e regras de negócio do nosso sistema
- **View**: parte estética, é a forma como os dados são apresentados ao utilizador.
- **Controller**: liga as diversas partes do sistema (o model e as views), controla a forma como manipulamos o software.

[Documentação MVC](#)

# A arquitectura MVC



O Front-End pede uma pizza:

1. O pedido é recebido nas rotas (routes.php) que vão indicar para onde é que ele vai.
2. O Controller indica as acções necessárias a concretizar o pedido, auxiliado pelo Model.
3. As Views processam e apresentam os resultados.

# Laravel e Composer



# Laravel - Instalação

- PHP
- mySql
- Server (incluído no Laravel, mas caso pretendam no futuro usar só PHP precisam do XAMPP, Laragon, Valet, etc)
- Composer

[Instalação](#)

# Laravel – Outras Ferramentas

- Chrome
- IDE – Visual Code, Php Storm, ..
- Terminal
- MySql Workbench



# Laravel – Extensões Visual Code

- PHP IntelliSense
- PHP Namespace Resolver
- Laravel-blade
- Laravel Blade Snippets
- Laravel goto view
  
- Laravel extension pack
- Laravel go-to-components
- Laravel Extra Intellisense

# o Composer

*php*



- gestor de dependências do php inspirado no npm do Node
- fornece recursos de carregamento automático de bibliotecas
- gere as dependências: instalação, actualização e remoção
- usa um ficheiro chamado composer.json

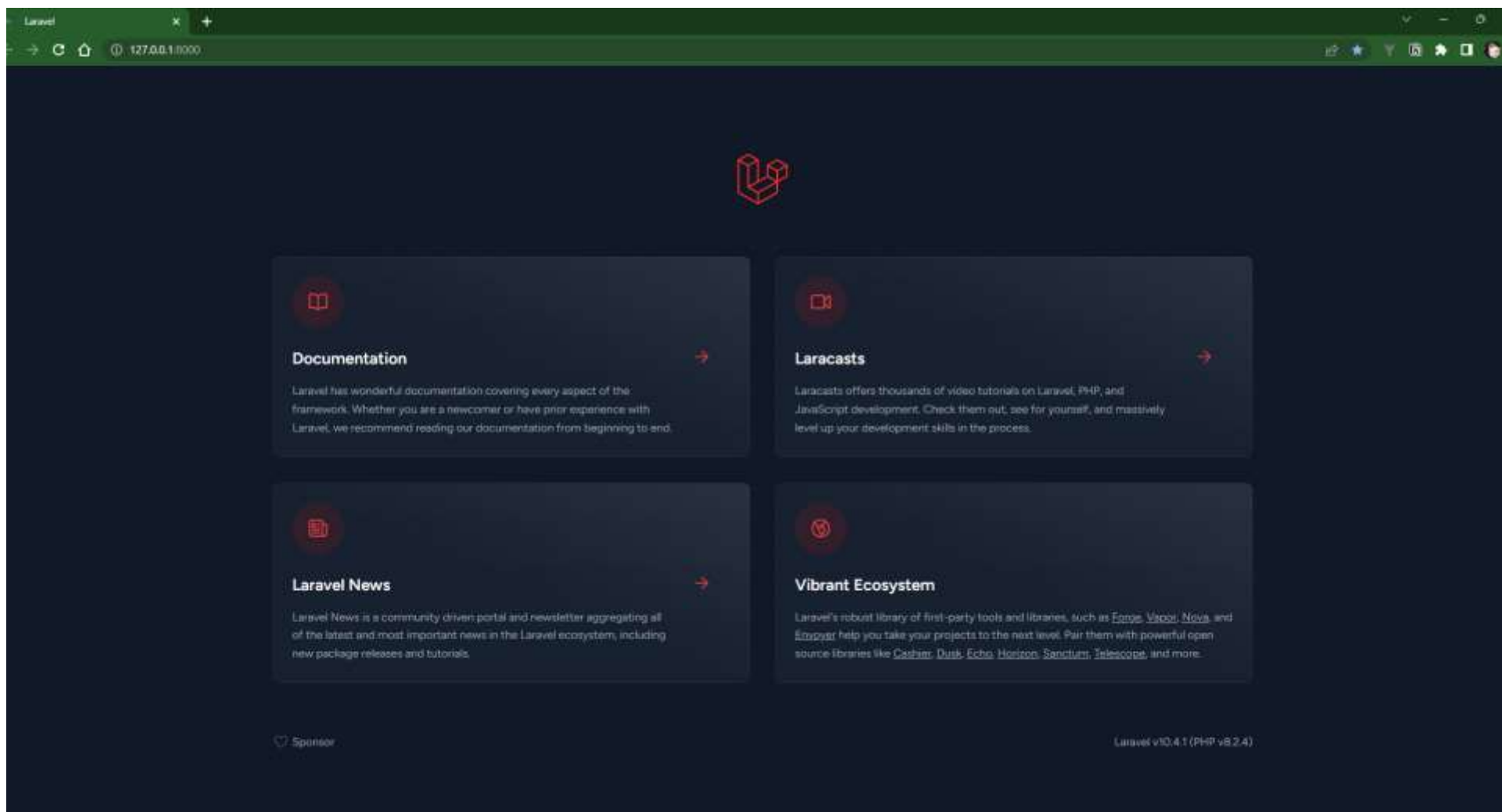
[documentação](#)

# Laravel – Instalação

**Nota:** instalar globalmente em Program Files

1. PHP: PHP  $\geq$  8.1 para a versão 10 do Laravel -> [Documentação](#)
2. [Instalar o Composer](#)
3. Abra o terminal e crie um novo projecto: `composer create-project Laravel/Laravel Web_ServerSide`
4. Abra o projecto e corra no terminal, na raíz do projecto: `php artisan serve`

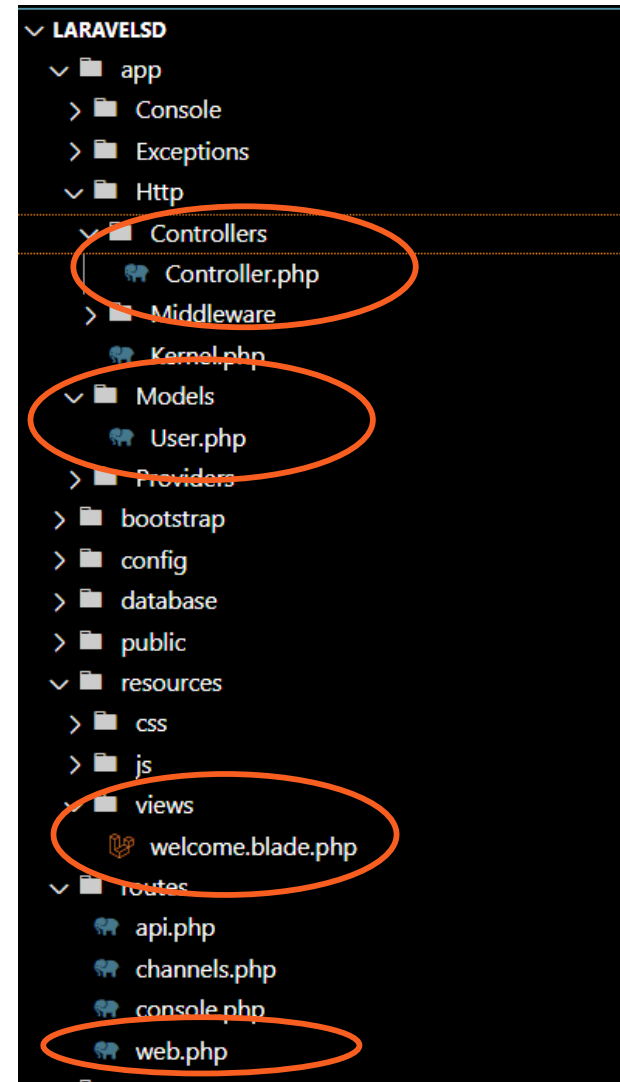
# Laravel



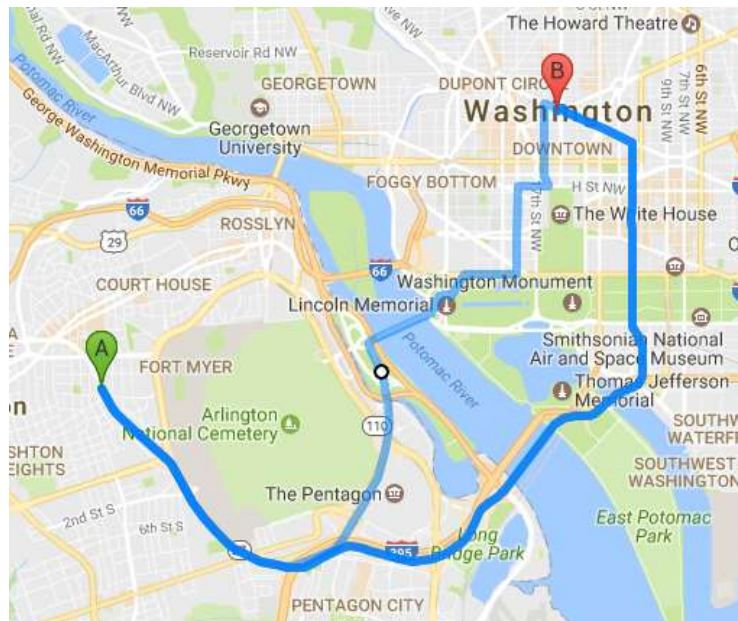
# Laravel – Comandos Iniciais e Estrutura

## No terminal:

- Php artisan serve -> correr o servidor
- Ctrl + C -> pára o servidor
- Php artisan -> ver todos os comandos
- Php artisan make:controller



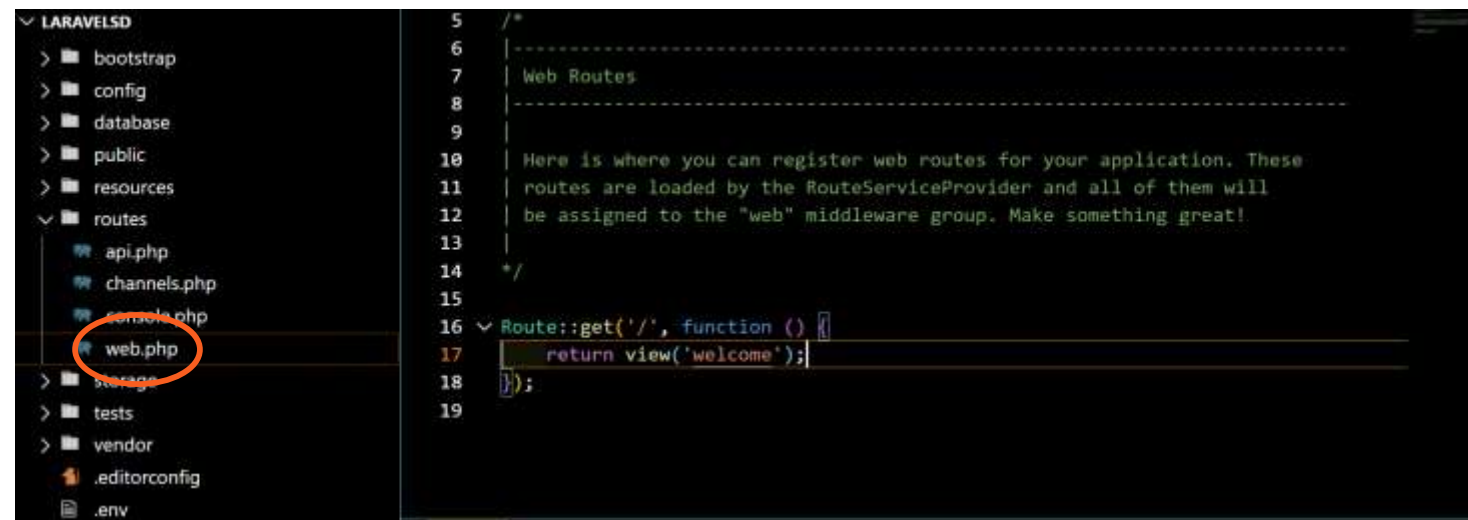
# Rotas





# As Rotas

- Ponto de entrada na aplicação
- Onde registamos os nossos 'caminhos'
- Para ver todas as rotas e sua informação: `php artisan route:list`

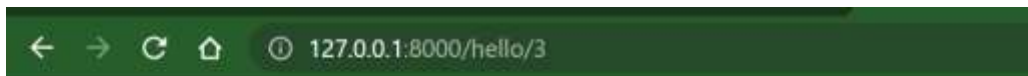


```
Route::get('/hello_world', function () {
    return "</h1>Hello World<h1>";
});
```

# Rotas com parâmetros

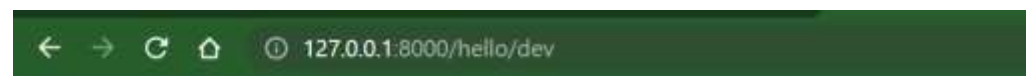
```
Route::get('/hello/{id}', function ($id) {  
    return '<h1>Hello</h1>'.$id;  
});
```

Nas rotas podemos definir parâmetros e que são carregados na página conforme haja ou não um valor.



Hello

3



Hello

dev

# Dar nomes às Rotas

As rotas podem ser definidas com nomes. A vantagem é que aquela rota fica sempre associada aquele nome. Caso tenhamos que mudar a sintaxe de como ela aparece ao utilizador ela muda automaticamente em todo o lado.

No exemplo a rota foi definida com o nome 'contacts.show' e é chamada através da route('contacts.show') quando quisermos direccionar para lá;

```
Route::get('/hello', function () {  
    return '<h1>Hello Turma Software Developer</h1>';  
})->name('contacts.show');
```

```
<a href="/portfolio/portfolio/">Portfolio</a>  
<a href=" . route('contacts.show') . ">Contactos</a>
```

# Rota Fallback

É a rota que é chamada quando o utilizador chama uma rota que não existe, ao invés de dar um erro 404 Not Found.

Usamos através da `Route::fallback` como no exemplo.

```
✓ Route::fallback(function () {  
    return '<h1>Ups, essa página não existe</h1>';  
});
```

# Views



# Views

- Na arquitectura MVC as Views são a camada que representa a interacção dos utilizadores com a nossa aplicação:
- Em Laravel as Views encontram-se no caminho: resources -> views.

Podemos criar dois tipos de views:

- Um ficheiro php normal
- Uma blade no formato my\_view.blade.php onde podemos usar todas as funcionalidades da Laravel Blade

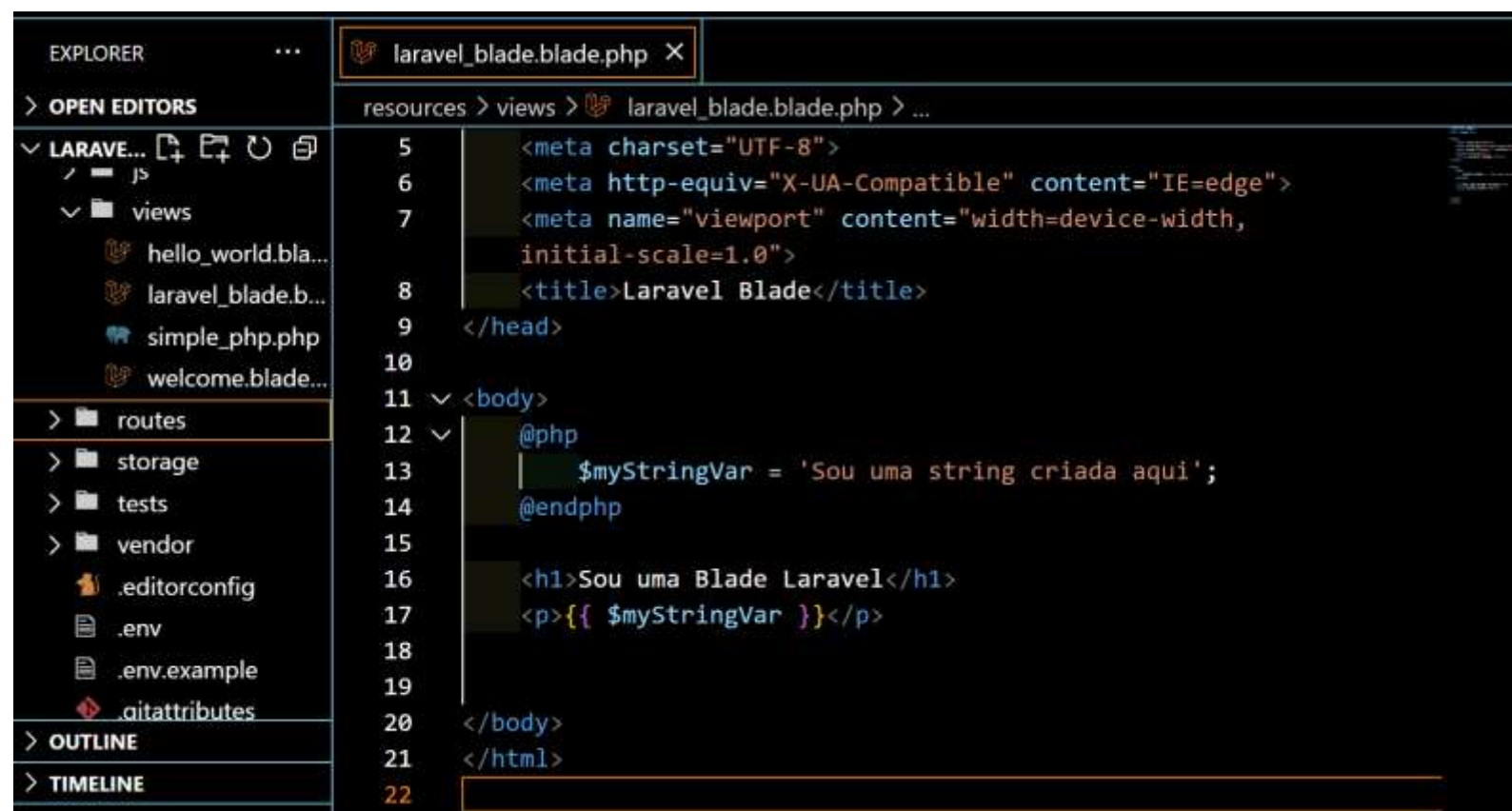
[Documentação](#)



# Views – a Blade do Laravel

Na Blade do Laravel podemos colocar:

- Html
- CSS
- JS
- Variáveis PHP
- Dados da Base de Dados
- ..



The screenshot shows a code editor with a sidebar on the left and a main editing area on the right. The sidebar has sections for 'EXPLORER', 'OPEN EDITORS', and 'OUTLINE'. The 'EXPLORER' section shows a file tree with folders like 'resources', 'views', 'routes', 'storage', 'tests', and 'vendor', and files like '.editorconfig', '.env', '.env.example', and '.gitattributes'. The 'OPEN EDITORS' section shows the current file 'laravel\_blade.blade.php' in the 'resources > views' directory. The main editing area shows the content of the file, which is a Blade template. The code includes meta tags for charset, http-equiv, and viewport, a title tag, and a body section. Inside the body, there is a PHP block that defines a variable and an HTML block that displays the variable's value.

```
5 <meta charset="UTF-8">
6 <meta http-equiv="X-UA-Compatible" content="IE=edge">
7 <meta name="viewport" content="width=device-width,
  initial-scale=1.0">
8 <title>Laravel Blade</title>
9 </head>
10
11 <body>
12   @php
13     $myStringVar = 'Sou uma string criada aqui';
14   @endphp
15
16   <h1>Sou uma Blade Laravel</h1>
17   <p>{{ $myStringVar }}</p>
18
19 </body>
20
21 </html>
22
```

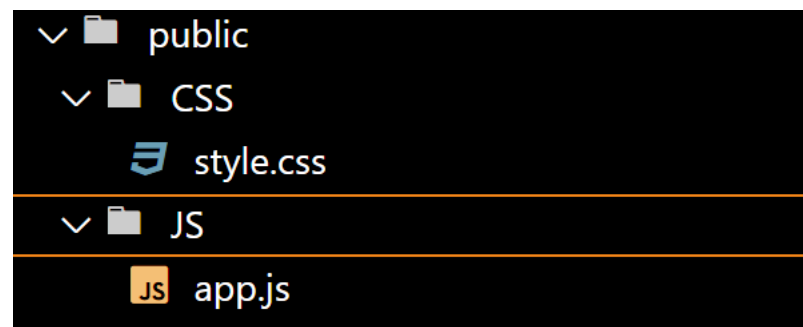
# Exercício



1. Crie uma rota para adicionar utilizadores e dê um nome à mesma.
2. Crie uma Blade que diga “Olá, aqui podes Adicionar Utilizadores” e associe à rota.
3. Utilizando a Blade Users\_Home já criada, acrescente na lista um item chamado “Adicionar Utilizador” .  
Ao clicar deverá ir para a Blade criada no ponto anterior.
4. Crie uma Blade de Fallback e associe à rota respectiva para mais tarde a podermos personalizar.

# Aplicar CSS e JS

- Para adicionar ficheiros JS e CSS às nossas Views, os mesmos devem ser criados na pasta public.



- O Laravel tem um Helper chamado [asset\(\)](#) que nos permite aceder ao caminho public sem nos preocuparmos com a estrutura de pastas ou em ter que mudar o caminho conforme a nossa localização.
- Podemos então adicionar estilo e JS ao nosso projecto desta forma.

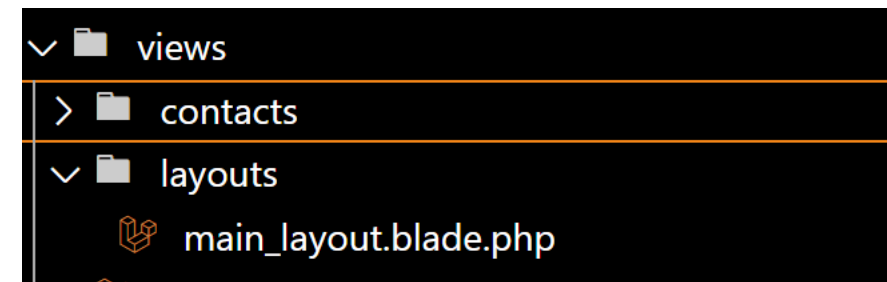
```
<link rel="stylesheet" href="{{ asset('CSS/style.css') }}">
<script src="{{ asset('JS/app.js') }}" defer></script>
```

# Criar um Layout Master

O Laravel permite-nos criar um layout 'Master' que depois podemos aplicar a várias páginas. Por exemplo, um menú e um rodapé comuns, estilo, JS, para não termos que o refazer em todas as blades.

Para criar e aplicar um Layout Master, seguimos os seguintes passos:

1. Criar nas Views uma pasta de Layouts onde criamos o nosso main e aí podemos colocar o que se vai repetir sempre: estrutura HTML, CSS, etc.



# Criar um Layout Master

2. Nesse ficheiro criar uma blade com tudo o que é reutilizável, por exemplo :
- um menú Bootstrap a ser aplicado em todas as blades

Devemos abrir um espaço para encaixar o conteúdo através do  
`@yield('content')`

```
<!-- Bootstrap CSS -->
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/
bootstrap@5.3.0-alpha2/dist/css/bootstrap.min.css">

<link rel="stylesheet" href="{{ asset('CSS/style.css') }}">

<!-- Bootstrap JS -->
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.
0-alpha2/dist/js/bootstrap.bundle.min.js"></script>

<script src="{{ asset('JS/app.js') }}" defer></script>
</head>

<body>
<nav class="navbar navbar-expand-lg bg-body-tertiary">...
</nav>
@yield('content')
</body>
```

# Criar um Layout Master

3. Na blade onde queremos aplicar o layout devemos deixar apenas o essencial que a diferencia.

No início da página estendemos o conteúdo do layout através de `extends('oNossoLayout')` e depois “encaixamos” o nosso conteúdo com `@section('content') / @endsection`

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

```
@section('content')
```

```
<div class="container">
```

```
<table class="table">...
```

```
</table>
```

```
</div>
```

```
@endsection
```



# Exercício: Layouts



Utilizando o Layout criado, aplique o mesmo a todas as blades que já temos.  
As Blades deverão ficar apenas com o essencial.

# Blade – Ifs e Loops

As Blades Laravel incluem a sua própria sintaxe para usar as funcionalidades do php.

Podemos usar na mesma PHP crú, mas os componentes do Laravel são mais eficientes e mantêm a aplicação limpa.

Existem para a blade if, while, for,.. Podem ver [aqui](#).

## Síntaxe PHP Crú

eu sou uma variável criada no PHP CRU

eu sou uma variável que existe

## Síntaxe Laravel

eu sou uma variável criada na Sintaxe da Blade

eu sou uma variável que existe

```
<!-- Síntaxe PHP Crú -->
<?php
$myPhpVar = 'eu sou uma variável criada no PHP CRU';
?>

<!-- Síntaxe Laravel -->
@php
    $myBladeVar = 'eu sou uma variável criada na Sintaxe da Blade';
    $testVar = 'eu sou uma variável que existe';
    $testNullVar = null;
@endphp

@section('content')
    <h1>Síntaxe PHP Crú</h1>
    <?php
    echo $myPhpVar . '</br>';
    if (!empty($testNullVar)) {
        echo $testNullVar;
    } else {
        echo $testVar;
    }
    ?>
    <br>

    <h1>Síntaxe Laravel</h1>
    {{ $myBladeVar }}
    <br>
    @if (!empty($testNullVar))
        {{ $testNullVar }}
    @else
        {{ $testVar }}
    @endif
    <h1>Sou Todos os Contactos</h1>
```

# Controllers



# Controllers

É onde está a lógica da aplicação: onde declaramos as variáveis, trabalhamos os dados, etc.

No Laravel encontramos os Controllers em app->Http->Controllers.

Podemos criar um novo Controller correndo no terminal o seguinte comando: `php artisan make:controller NossoNomeController`

Podemos também adicionar manualmente um ficheiro php, mas ao usar os comandos Laravel os ficheiros são criados com a estrutura base certa.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Foundation\Auth\Access\AuthorizesRequests;
use Illuminate\Foundation\Validation\ValidatesRequests;
use Illuminate\Routing\Controller as BaseController;

class Controller extends BaseController
{
    use AuthorizesRequests, ValidatesRequests;
}
```

[Documentação](#)

# Controllers

Até agora temos estado a chamar as Views directamente nas rotas através de uma função.. Uma vez que uma função contém lógica de código, a mesma deverá passar para um Controller. Na rota deverá constar a seguinte estrutura:

```
Route::get('/home_contacts', [HomeController::class, 'index'])->name('index.contacts');
```



No cimo da página de Routes temos que declarar o Controller que estamos a usar.

```
1  <?php
2
3  use App\Http\Controllers\HomeController;
4  use Illuminate\Support\Facades\Route;
5
```

# Controllers

No Controller criamos uma função index e aí retornamos a View que queremos.

Estas funções devem ser públicas para poderem ser acedidas por toda a aplicação (neste caso nas rotas).

```
class HomeController extends Controller
{
    public function index()
    {
        return view('contacts.home_contacts');
    }
}
```

# Exercício: da Rota para o Controller



Centro para o Desenvolvimento  
de Competências Digitais

Usando o nosso UserController crie funções com as Views usadas anteriormente.

**Nota:** as funções deverão ter nomes auto-explicativos e estar escritas em camelCase.

# Retornar uma View com Dados

Uma vez que é no Controller que se concentra a lógica da nossa aplicação, será aí que vamos manipular os dados antes de aparecerem na View.

Podemos por exemplo criar um array de dados e carregar o mesmo com a View.

```
//função pública
public function index()
{
    $myVar = 'Sou uma variável a ser enviada para a Blade';

    $contactInfo = [
        'name' => 'Nome da Pessoa',
        'phone' => 'Contacto da Pessoa'
    ];

    //retornar a view com dados
    return view('contacts.home_contacts', compact('myVar',
        'contactInfo'));
}
```



# Retornar uma View com Dados

Na Blade podemos chamar então chamar os dados utilizando os helpers da Blade.

```
<h2>{{ $myVar }}</h2>

<h2>Informação a colocar nos Contactos</h2>
<ul>
    <li>{{ $contactInfo['name'] }}</li>
    <li>{{ $contactInfo['phone'] }}</li>
</ul>
```

**Sou uma variável a ser enviada para a Blade**

**Informação a colocar nos Contactos**

- Nome da Pessoa
- Contacto da Pessoa

# Exercício:

## Views com dados



Centro para o Desenvolvimento  
de Competências Digitais

1.No UserController crie uma função protegida com o nome getCesaeInfo e defina dentro dela o seguinte array:

```
$cesaeInfo = [  
    'name' => 'Cesae',  
    'address' => 'Rua Ciríaco Cardoso 186, 4150-212 Porto',  
    'email' => 'cesae@cesae.pt'  
];
```

2.Na função onde está a carregar a View all\_users chame esta função de modo a ir buscar os dados.

Carregue a View com os dados.

3. Na Blade chame os dados através do helper do Laravel. Ex: {{ \$cesaeInfo[ 'name' ] }}.

# Retornar uma View com Array

Para trabalhar melhor os dados e manter o código organizado de forma a reutilizá-lo podemos criar uma função protegida ou privada (só podemos aceder através do Controller) e nela criar um array de Contactos.

```
//função que só se acede neste Controller
protected function getContacts()
{
    $contacts = [
        ['id' => 1, 'name' => 'Sara', 'phone' => '985654455'],
        ['id' => 2, 'name' => 'Bruno', 'phone' => '985654455'],
        ['id' => 3, 'name' => 'Márcia', 'phone' => '985654455']
    ];

    return $contacts;
}
```

# Retornar uma View com Array

Podemos aceder às funções dentro do nosso Controller através do `$this->aNossafunção()` e colocar os dados dentro de uma variável que depois enviamos para ser lida na View.

```
//função pública
public function allContacts()
{
    //chamar a função do nosso Controller
    $contacts = $this->getContacts();

    //retornar a view com os dados dos contactos
    return view('contacts.all_contacts', compact('contacts'));
}
```

# Retornar uma View com Array

```
<tbody>
  @foreach ($contacts as $item)
    <tr>
      <td>{{ $item['id'] }}</td>
      <td>{{ $item['name'] }}</td>
      <td>{{ $item['phone'] }}</td>
    </tr>
  @endforeach
</tbody>
```

Na Blade podemos usar um ciclo For e chamar assim os nossos dados

# Controllers com Recursos

Numa aplicação real iremos precisar de funções para fazer o CRUD (Create, Read, Update and Delete) e outras recorrentes.

Para simplificar o processo, em Laravel podemos criar um Controller já com estas funções, acrescentando um -r no fim do comando que gera o controller:

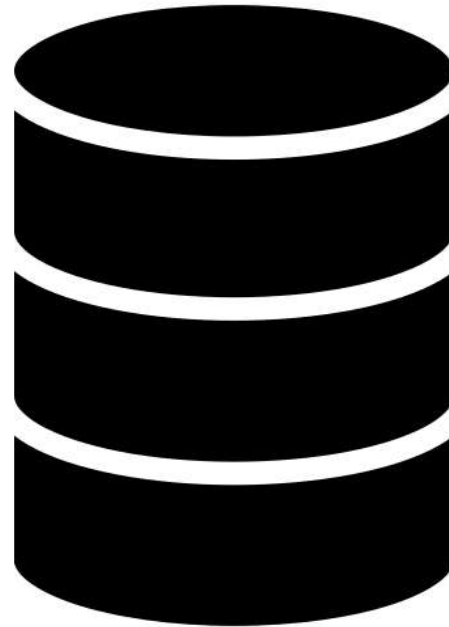
Ex: `php artisan make:controller  
CompanyController -r`

```
    * Display the specified resource.
    */
    public function show(string $id)
    {
        //
    }

    /**
     * Show the form for editing the specified resource.
     */
    public function edit(string $id)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $request, string $id)
```

# Bases de Dados e Migrações



# Bases de Dados

## [Documentação](#)

O Laravel contém um leque enorme de funcionalidades relacionadas com Bases de Dados, nomeadamente:

- Migrations
- Seeding
- Query Builder
- Eloquent
- ..

Para as usarmos precisamos de:

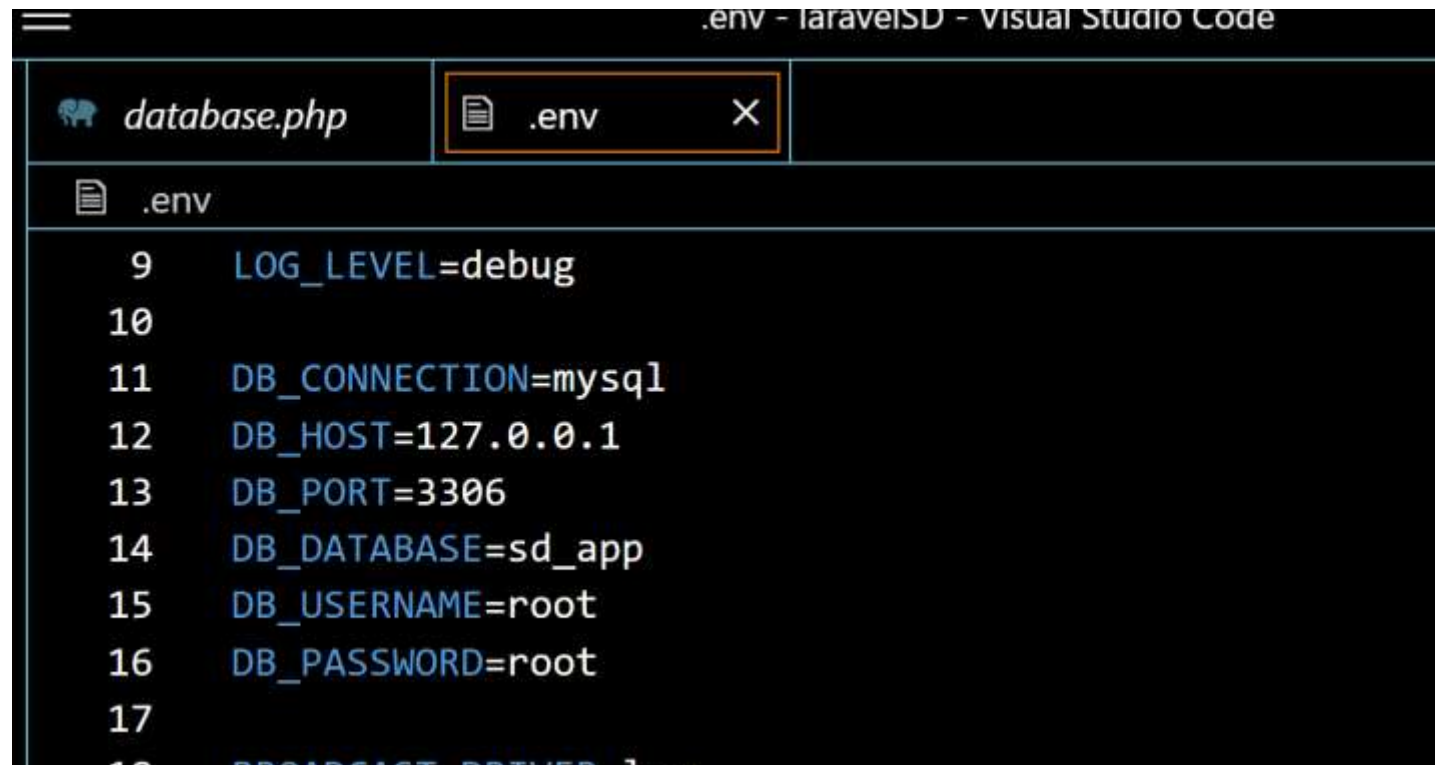
- 1 Mysql: `mysql --version`
- 2 Entrar como root no Mysql: `mysql -u root` ou `mysql -u root -p`
- 3 Criar uma base de Dados: `CREATE DATABASE server-side;`



# Bases de Dados: Configurações

As configurações da Base de Dados encontram-se em: config/database.php e este ficheiro remete para o .env, que devemos alterar.

Após termos alterado para os nossos dados, corremos no terminal: php artisan migrate. Se tudo estiver ok o Laravel irá criar tabelas predefinidas.



```
.env - laravelSD - Visual Studio Code

database.php .env X

.env

9  LOG_LEVEL=debug
10
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=sd_app
15 DB_USERNAME=root
16 DB_PASSWORD=root
17
18 BROADCAST_DRIVER=log
```

```
2019_08_19_000000_create_failed_jobs_table ..... 49ms DONE
2019_12_14_000001_create_personal_access_tokens_table .... 41ms DONE
```

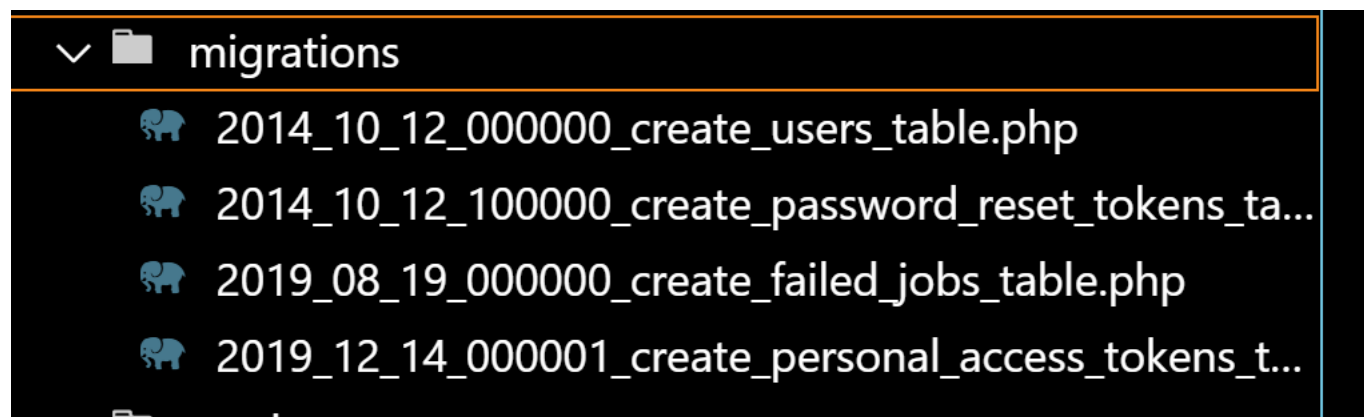
# Bases de Dados: As Migrações

As migrações são como uma Versão de controlo para a nossa base de dados, como se fosse o código fonte da Base de Dados.

Ao longo do tempo as Bases de Dados vão sendo alteradas. Se trabalharmos em equipa, bastará dizer à equipa para correr o migrate através do código fonte e automaticamente as Tabelas são actualizadas.

As migrações encontram-se em database\migrations

[Documentação](#)



# Bases de Dados: As Migrações

```
/* Run the migrations.
*/
public function up(): void
{
    Schema::create('password_reset_tokens', function (Blueprint $table) {
        $table->string('email')->primary();
        $table->string('token');
        $table->timestamp('created_at')->nullable();
    });
}

/**
 * Reverse the migrations.
 */
public function down(): void
{
    Schema::dropIfExists('password_reset_tokens');
}
```

- Podemos ver o estado das migrações com: **php artisan migrate:status**
- Podemos criar uma migração com: **php artisan make:migration nomeDaMigração**
- Podemos fazer rollback das migrações com: **php artisan migrate:rollback**

(ver [documentação](#) para naming correcto)

# As Migrações : criar uma Tabela

Podemos criar tabela através de uma Migração correndo o seguinte comando:

Desta forma a tabela vem com duas colunas predefinidas:

- `$table->id();`  
    tipo `unsignedBigInteger` // PRIMARY KEY // `AUTO_INCREMENT`
- `$table->timestamps();`  
    duas colunas: `created_at` and `updated_at` como timestamps

```
Run the migrations:
*/
public function up(): void
{
    Schema::create('flights', function (Blueprint $table) {
        $table->id();
        $table->timestamps();
    });
}

/**
 * Reverse the migrations.
 */
```

# As Migrações: definir colunas

Da mesma forma que fazemos no mySQL tradicional, nas migrações podemos também criar colunas com [vários tipos de dados](#).

```

/**
 *
 * public function up(): void
 * {
 *     Schema::create('flights', function (Blueprint $table) {
 *         $table->id();
 *         $table->string('name');
 *         $table->integer('phone')->nullable();
 *         $table->timestamps();
 *     });
 * }
 */

```

COMMENTS:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	BIGINT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
name	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
phone	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
created_at	TIMESTAMP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
updated_at	TIMESTAMP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

# As Migrações: rollback

Caso queiramos reverter a migração podemos sempre correr o comando php migrate:rollback e ele reverte a migração para o que tivermos na função down.

```
/**
 * Reverse the migrations.
 */
public function down(): void
{
    Schema::dropIfExists('flights');
}
```

# As Migrações : criar uma Tabela

Se correremos php artisan migrate ele criará a tabela na nossa Base de Dados.

Info	Columns	Indexes	Triggers	Foreign keys	Partitions	Grants	DDL			
Column	Type	Default Value	Nullable	Character Set	Collation	Privileges	Extra	Comments		
◇ id	bigintunsigned		NO			select,insert,update,references	auto_increment			
◇ created_at	timestamp		YES			select,insert,update,references				
◇ updated_at	timestamp		YES			select,insert,update,references				

# As Migrações: alterar uma Tabela

Sempre que houver alterações a uma tabela depois da criação original (adicionar ou remover colunas, etc.) devemos criar um novo ficheiro de migrações.

Desta forma mantemos a coerência caso trabalhem com outros membros numa equipa e evitamos corromper dados.

O nome da migração deverá seguir o seguinte esquema:

add\_column\_to\_tablename\_table.

Exemplo: php artisan make:migration

add\_address\_to\_users\_table

```
    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('address');
        });
    }
}
```

**Nota:** não esquecer de criar o down()!



# Migrações – Exercício



1. Usando as Migrações do Laravel e as indicações da documentação, crie uma tabela chamada Tasks com as seguintes colunas:
  - name, string, not Nullable
  - description, text, nullable
  - due\_at, date, nullable
  - status, boolean, nullable
2. Corra a migração e verifique se a tabela foi correctamente criada na Base de Dados

# As Migrações: Chaves Estrangeiras

Nas migrações podemos também construir tabelas relacionais, por exemplo, ligar uma tabela de Tasks à de Users, da [seguinte forma](#):

```
* Run the migrations.
*/
public function up(): void
{
    Schema::table('tasks', function (Blueprint $table) {
        $table->unsignedBigInteger('user_id')->after('id');
        $table->foreign('user_id')->references('id')->on('users');
    });
}
```

## Notas:

- o tipo de dados da FK tem que ser igual ao da PK!
- Idealmente devemos adicionar as FK logo na criação para não corromper ligações

# Operações à Base de Dados

O Laravel oferece duas formas para se fazer operações à Base de Dados:

- [Query Builder](#)
- [Eloquent](#) (usando os Models)

# Query Builder: Insert e Update

Para fazermos operações com o Query Builder usaremos o Facade DB :  
`use Illuminate\Support\Facades\DB;`

```
namespace App\Http\Controllers;  
use Illuminate\Support\Facades\DB;
```

```
DB::table('users')  
->insert([  
    'name' => 'Sara',  
    'email' => 'Sara@gmail.com',  
    'password' => 'Sara1234',  
]);
```

```
DB::table('users')  
->where('id', 1)  
->update([  
    'email_verified_at' => now()  
]);
```

# Query Builder: updateOrInsert e Delete

```
DB::table('users')
    ->updateOrInsert(
        [
            'email' => 'Sara@gmail.com'
        ],
        [
            'email_verified_at' => now()
        ]
    );
```

O método `updateOrInsert` verifica se existe alguma linha com o primeiro argumento (neste caso com `email = 'Sara@gmail.com'`).

Se existir, coloca a verificação para `now`, se não existir cria esse user com email `'Sara@gmail.com'` e verificado agora.

```
DB::table('users')
    ->where('email', 'Sara@gmail.com')
    ->delete();
```

# Query Builder: receber resultados da BD

Receber todos os Users



```
DB::table('users')  
->get();
```

```
<tbody>  
  @foreach ($contacts as $item)  
    <tr>  
      <td>{{ $item->id }}</td>  
      <td>{{ $item->name }}</td>  
      <td>{{ $item->email }}</td>  
    </tr>  
  @endforeach  
</tbody>  
</table>
```

O que é retornado da BD é um objecto e não um array. Logo, chamaremos os dados do objecto com \$item->anossacoluna

Receber um User específico



```
DB::table('users')  
->where('email', 'Sara@gmail.com')  
->first();
```

# Query Builder – Exercício



Centro para o Desenvolvimento  
de Competências Digitais

1. Na nossa View Home, adicione um item chamado: 'todas as tarefas'. O item deverá abrir uma view com uma tabela Bootstrap.

## Notas de Ajuda:

1 - criar a rota com um name, criar uma função pública `allTasks` no Controller, criar uma view `all_tasks` e chamá-la na função.

2 – na view home adicionar um li igual aos anteriores e chamar a rota criada usando o `{{route('nome da nova rota')}}`

2. Na tabela de tarefas, criar dummy content de tasks (pode ser manualmente).

No Controller que estamos a usar, criar uma função protegida chamada `getAllTasks` que aceda à base de dados e retorne todas as tarefas criadas.

3. Na função onde carregamos a view de todas as tarefas, chamar a função `getAllTasks` e enviá-la para a View.

4. Na Blade colocar na tabela o nome da tarefa, o estado e a data de conclusão.

Ponto extra: na query das tasks fazer um join que nos traga também os users e adicionar no fim da tabela uma coluna chamada: **pessoa responsável**.

# Modelos

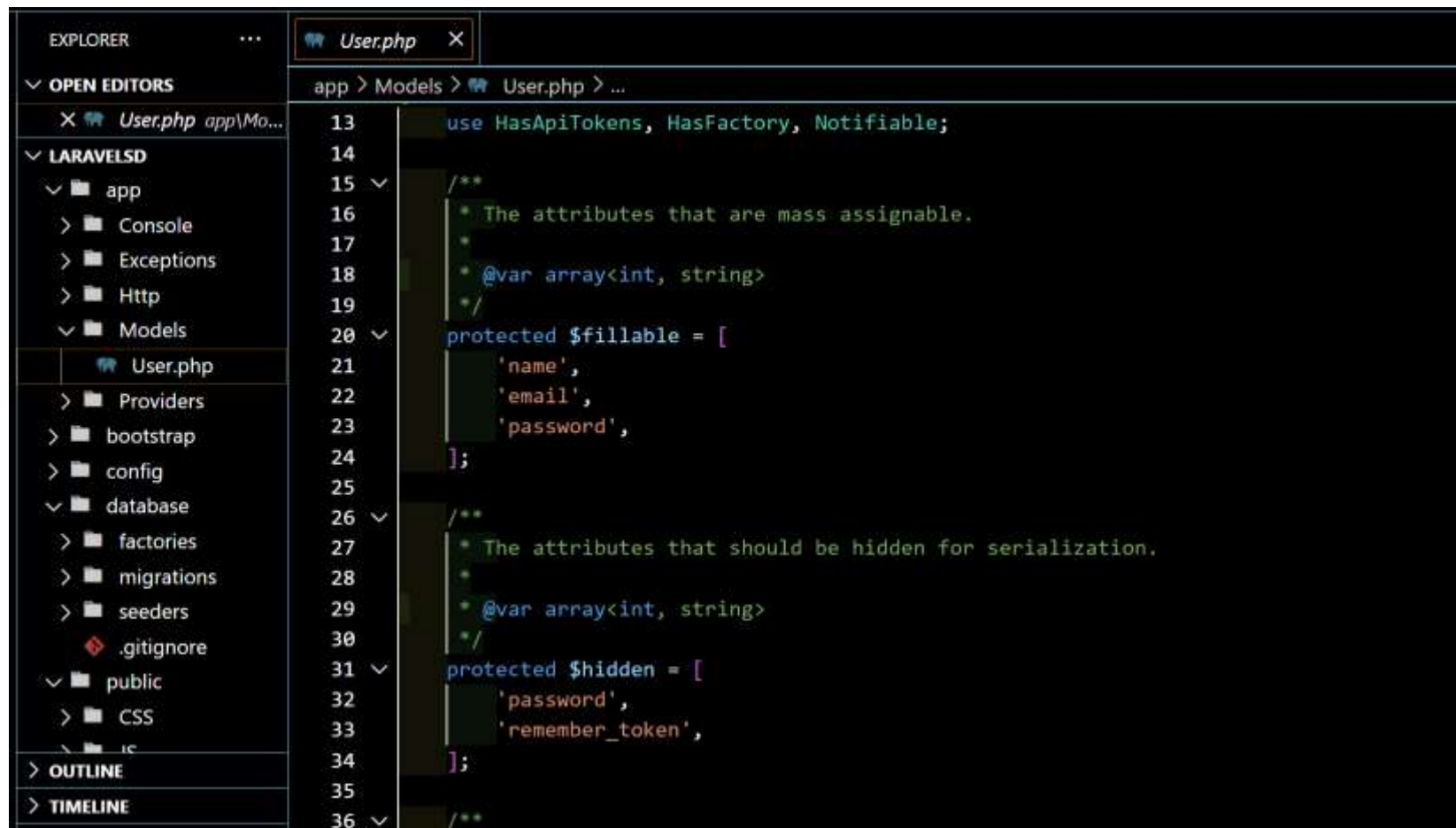




# Modelos

- Representam uma tabela da Base de Dados e permitem interagir com a mesma: retornar dados, inserir, actualizar, etc..
- Localizam-se em `app\Models` e já existe um de `Users` por defeito.

[Documentação](#)



The screenshot shows an IDE with the Explorer panel on the left displaying the project structure. The 'app' directory is expanded, showing 'Models' and 'User.php'. The main editor displays the code for 'User.php' with line numbers 13 to 36. The code includes imports for 'HasApiTokens', 'HasFactory', and 'Notifiable'. It also contains two docblocks: one for mass assignable attributes and another for attributes to be hidden during serialization.

```
13 use HasApiTokens, HasFactory, Notifiable;
14
15 /**
16  * The attributes that are mass assignable.
17  *
18  * @var array<int, string>
19  */
20 protected $fillable = [
21     'name',
22     'email',
23     'password',
24 ];
25
26 /**
27  * The attributes that should be hidden for serialization.
28  *
29  * @var array<int, string>
30  */
31 protected $hidden = [
32     'password',
33     'remember_token',
34 ];
35
36 /**
```

# Modelos

- Para criar um modelo corremos o comando: `php artisan make:model oNossoNome`
- Para oNossoNome usamos as seguintes convenções:
  - ❖ O nome da nossa tabela no singular
  - ❖ Primeira letra em Maiúscula
  - ❖ Ex: tabela users, o nome do Model é User
- Se seguirmos as convenções, o Model automaticamente associa à nossa tabela o nome e a chave primária id.
- Podemos também criar um modelo e ao mesmo tempo criar uma migração para a tabela correspondente, correndo: `php artisan make:model Note -m`  
(neste caso criou a migração para a tabela Notes e o Model Note).

# Model – Exercício



1. Criar um Model para a tabela Tasks.

# Queries usando o Eloquent: retornar dados

1. Para usar as [queries do Modelo](#) temos que importar o modelo em questão no Controller: use `App\Models\oNossoModelo`;

```
//retornar todos da tabela flights  
Flight::all();
```

Method	Description
whereBetween, orWhereBetween	verifies that a column's value is between two values
whereNotBetween, orWhereNotBetween	verifies that a column's value lies outside of two values
whereIn, whereNotIn, orWhereIn, orWhereNotIn	verifies that a given column's value is contained within the given array
whereNull, whereNotNull, orWhereNull, orWhereNotNull	verifies that the value of the given column is <b>NULL</b> , not <b>NULL</b>
whereDate, whereMonth, whereDay, whereYear, whereTime	compare a column's value against a date, month, etc

```
//retornar o voo com id 1 da tabela flights  
Flight::where('id', 1)->first();
```

# Queries usando o Eloquent: Insert, Update and Delete

```
//inserir um User
User::insert([
    'name' => 'Márcia',
    'email' => 'Marcia@gmail.com',
    'password' => 'Marcia123',
]);
```

```
//atualizar ou criar
User::updateOrCreate(
    ['email' => 'Marcia@gmail.com'],
    [
        'name' => 'Márcia',
        'password' => 'Marcia1234'
    ]
);
```

```
//fazer update a um User
User::where('email', 'Marcia@gmail.com')
->update(['password' => 'Marcia2023']);
```

```
//apagar um User
User::where('email', 'Marcia@gmail.com')
->delete();
```

# Construir Views Dinâmicas



# Construir Views dinâmicas

As views dinâmicas são criadas de forma a que a tabela esteja sempre actualizada com os dados da BD e que possamos geri-la através de botões.

## Users

#	Nome	Email		
1	Sara	sara@gmail.com	Ver	Apagar
2	Bruno	Bruno@gmail.com	Ver	Apagar
3	Hélder	Hélder@gmail.com	Ver	Apagar
4	Ana	Ana@gmail.com	Ver	Apagar
5	Marcia	Marcia@gmail.com	Ver	Apagar

# Construir Views dinâmicas

Os botões irão encaminhar para rotas com o \$id que queremos manipular para que se processem as operações correspondentes.

```
<td>
  <a href="{{ route('view_contact', $item->id) }}"> <button
    type="button"
    class="btn btn-info">Ver</button></a>
  <a
    href="{{ route('delete_contact', $item->id) }}">
    <button type="button" class="btn btn-danger">Apagar</
    button></a>
</td>
```

```
Route::get('/view_contact/{id}', [HomeController::class, 'viewContact'])->name
('view_contact');
Route::get('/delete_contact/{id}', [HomeController::class, 'deleteContact'])->name
('delete_contact');
```



# Construir Views dinâmicas

Nas funções iremos ter o \$id e podemos manipular a base de dados conforme o user solicitou.

```
public function viewContact($id)
{
    $ourUser = User::where('id', $id)->first();

    //retornar a view com os dados do nosso User
    return view('contacts.view_contact', compact('ourUser'));
}

public function deleteContact($id)
{
    User::where('id', $id)->delete();

    return back();
}
```

# Tabelas Dinâmicas

## – Exercício



1. À semelhança do que fizemos nos Utilizadores, na Tabela onde temos todas as Tarefas acrescentar a cada tarefa um botão de Ver e outro de Apagar que cumpram as respectivas funções.

# Recursos

- [Documentação Laravel](#)
- [Laracasts](#)