

# Programação para Web com Frameworks

Desenvolvimento Web Full Stack

# PDO – PHP Data Objects

# PDO – PHP Data Objects

Unifica o acesso às diferentes extensões de bancos de dados presentes em PHP;

Provê uma API limpa e consistente, unificando a maioria das características presentes nas extensões de acesso a banco de dados

Unifica a chamada de métodos, graças à orientação a objetos a partir do **PHP 5**;

# PDO – PHP Data Objects

Sequência de passos:

1. A aplicação estabelece uma conexão com o SGBD;
2. A aplicação seleciona uma base de dados a ser utilizada durante a conexão;
3. A aplicação executa uma consulta específica (seleção, inserção, remoção...);
4. A aplicação fecha a conexão corrente com o SGBD;

# PDO – PHP Data Objects

PDO – Conexão com bancos de dados:

| Banco    | String de Conexão   |
|----------|---|
| SQLite   | <code>new PDO('sqlite: teste.db');</code>   |
| FireBird | <code>new PDO("firebird:dbname=C:\\base.GDB", "SYSDBA", "masterkey");</code>            |
| MySQL    | <b><code>new PDO('mysql:host=localhost;dbname=base', 'login', 'senha');</code></b>      |
| Postgres | <code>new PDO('pgsql:dbname=example; user=user; password=senha;host=localhost');</code> |

# PDO – Exemplos

# PDO – Exemplos

No **phpmyadmin**, configure o banco de dados da atividade:

- Database: **livros**
- Tabela: **famosos**
  - Campos: **id** (int, auto increment, chave primária) e **nome** (varchar[100]);

# PDO – Exemplos

## Exemplo 1: Inserindo dados via PDO

```
<?php

    $conn = new PDO('mysql:host=localhost;dbname=livros',
'root', '');

    $conn->exec("INSERT INTO famosos (nome) VALUES ('Thyago
Maia')");

    $conn = null;

?>
```

# PDO – Exemplos

## Exemplo 2: Listagem via PDO

```
<?php
    $conn = new PDO('mysql:host=localhost;dbname=livros',
'root', '');

$result = $conn->query("SELECT id, nome from famosos");

if($result)
    foreach($result as $row)
        echo $row['id'] . '-' . $row['nome'] . '<br>';

$conn = null;
?>
```

# PDO – Exemplos

## Exemplo 3: Excluindo dados via PDO

```
<?php

$conn = new
PDO('mysql:host=localhost;dbname=livros', 'root',
'');

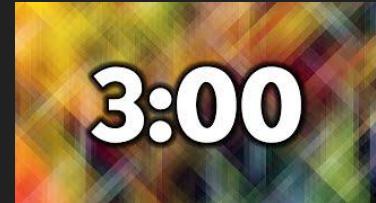
$conn->exec("DELETE FROM famosos WHERE id = 1");

$conn = null;

?>
```

# Exercício

# Exercício



## Etapa 1 – Preparação do Banco de Dados no **phpmyadmin**

Database: **sistema**

- Tabela: **cadastro**
  - Campos: **codigo** (int, auto increment, chave primária), **nome** (varchar[100]), **telefone** (varchar[100]) e **email** (varchar[100]);

# Exercício

## Etapa 2 – Preparação do Domínio

- Crie uma subpasta na pasta definida por você para as atividades da pós, chamada **sistema**;

# Exercício



## Etapa 3 – Criação de um formulário HTML

- Crie um formulário HTML que permita ao usuário digitar seu nome, telefone e e-mail;
  - Método de envio: POST;
  - Os dados deverão ser submetidos para o script “cadastrar.php”;
  - Salve o documento HTML na pasta sistema com o nome index.html;

# Exercício



Etapa 4 – Criação do script **cadastrar.php** na pasta **sistema**

- O script deverá inicialmente receber os dados submetidos na etapa anterior;
- Em seguida, via PDO, o script deverá inserir os dados no banco de dados;
- O script deverá **redirecionar o usuário** para o script **lista\_usuarios.php**;

# Exercício

Etapa 5 – Criação do script **lista\_usuarios.php** na pasta **sistema**

- O script deverá exibir todos os cadastros efetuados até o momento, além de disponibilizar um link que permita ao usuário efetuar um novo cadastro;

Etapa 6 – Testar a aplicação



# Exercício

# Exercício

## Etapa 1 – Preparação do Banco de Dados no **phpmyadmin**

- Utilizaremos a mesma base criada no exercício anterior;

# Exercício

## Etapa 2 – Preparação do Domínio

- Utilizaremos a pasta **sistema** criada anteriormente;

# Exercício

## Etapa 3 – Criação de um formulário HTML

- Utilizaremos o mesmo formulário do exercício anterior;



# Exercício

Etapa 4 – Criação da classe **Cadastro.php** na pasta **sistema/classes**

| Cadastro                              |  |
|---------------------------------------|--|
| -nome: string                         |  |
| -telefone: string                     |  |
| -email: string                        |  |
| -servidor: PDO                        |  |
| +setNome(nome: string) : void         |  |
| +setTelefone(telefone: string) : void |  |
| +setEmail(email: string) : void       |  |
| +getNome() : string                   |  |
| +getTelefone() : string               |  |
| +getEmail() : string                  |  |
| +inserir() : void                     |  |
| +exibir() :void                       |  |



# Exercício

## Etapa 5 – Atualização do script **cadastrar.php** da pasta **sistema**

- O script deverá:
  - Receber os dados submetidos na etapa anterior;
  - Instanciar um objeto da classe **Cadastro**;
  - Atribuir os dados recebidos para o objeto instanciado;
  - Executar o método **inserir** a partir do objeto instanciado;
  - Executar o método **exibir** a partir do objeto instanciado;
  - Disponibilizar um link que permita ao usuário efetuar um novo cadastro;

## Etapa 6 – Testar a aplicação.

Criando consultas preparadas (Prepared Statements) via PDO

# Prepared Statements

Uma “consulta preparada” é um recurso disponível na classe PDO que permite pré-carregar consultas SQL que serão executadas repetidamente;

Nelas, será necessário apenas indicar possíveis parâmetros de consulta que irão variar de acordo com cada execução;

Dessa forma, reduzimos tempo de parsing e preparações de consultas (a consulta preparada não precisa ser pré-carregada antes de cada execução);

Também são muito úteis para evitar SQL Injections, já que os parâmetros de consultas preparadas já serão “escapadas” pela classe PDO.

## Exemplo 1: Inserindo dados via PDO a partir de uma consulta preparada

```
<?php
$conn = new PDO('mysql:host=localhost;dbname=livros', 'root', '');
$query = $conn->prepare("INSERT INTO famosos (nome) VALUES (:nome)");
$query->bindParam(':nome', $nome);
$nome = 'Thyago Maia';
$query->execute();
$nome = 'Theo Maia';
$query->execute();

$conn = null;
?>
```

## Exemplo 2: Selecionando dados via PDO a partir de uma consulta preparada

```
<?php
$conn = new PDO('mysql:host=localhost;dbname=livros', 'root', '');

$query = $conn->prepare("SELECT * FROM famosos");
$query->execute();

$tabela = $query->fetchAll(PDO::FETCH_ASSOC);

foreach($tabela as $linha) {
    echo "{$linha['id']} - {$linha['nome']}<br/>";
}

$conn = null;
?>
```

# Componentes PHP

# Componentes PHP

Pacotes de código que ajudam a resolver problemas específicos em sua aplicação PHP;

- Para que “reinventar a roda”, caso exista um componente PHP que atenda algum requisito de projeto?

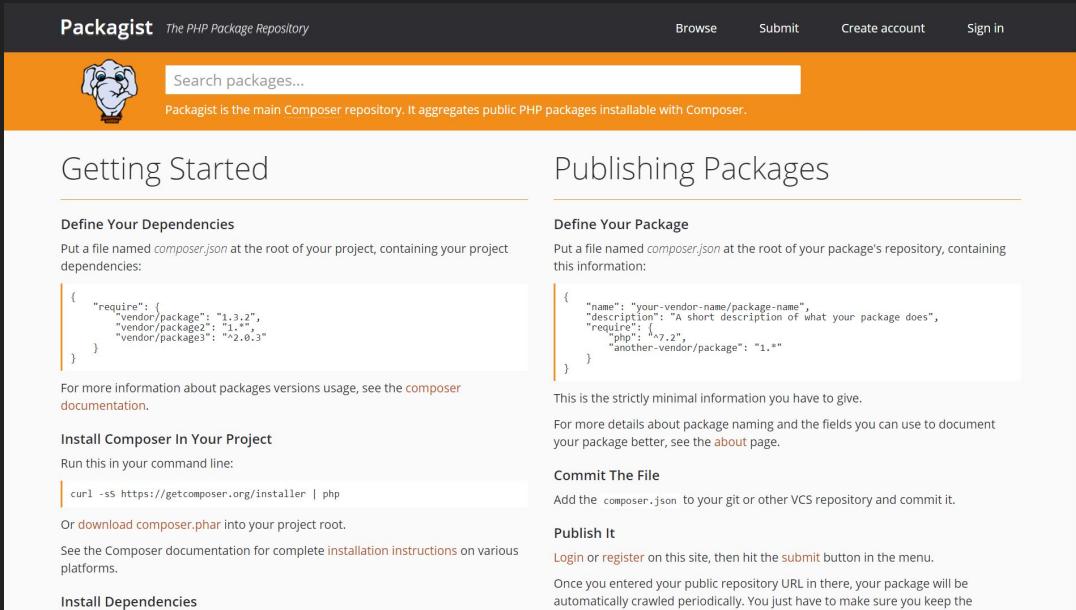
Em vez de recriar uma funcionalidade já implementada, utilizamos componentes PHP;

Assim, gastamos mais tempo resolvendo os objetivos mais amplos do nosso projeto.

# Componentes PHP

## Encontrando componentes PHP

- Packagist (<https://packagist.org/>)



The screenshot shows the Packagist website, "The PHP Package Repository". The top navigation bar includes links for "Browse", "Submit", "Create account", and "Sign in". The main content area is divided into two main sections: "Getting Started" on the left and "Publishing Packages" on the right.

**Getting Started**

### Define Your Dependencies

Put a file named `composer.json` at the root of your project, containing your project dependencies:

```
{ "require": { "vendor/package": "1.3.2", "vendor/package2": "1.1", "vendor/package3": "2.0.3" } }
```

For more information about packages versions usage, see the [composer documentation](#).

### Install Composer In Your Project

Run this in your command line:

```
curl -sS https://getcomposer.org/installer | php
```

Or [download composer.phar](#) into your project root.

See the Composer documentation for complete [installation instructions](#) on various platforms.

### Install Dependencies

**Publishing Packages**

### Define Your Package

Put a file named `composer.json` at the root of your package's repository, containing this information:

```
{ "name": "your-vendor-name/package-name", "description": "A short description of what your package does", "require": { "php": "7.2", "another-vendor/package": "1.*" } }
```

This is the strictly minimal information you have to give.

For more details about package naming and the fields you can use to document your package better, see the [about](#) page.

### Commit The File

Add the `composer.json` to your git or other VCS repository and commit it.

### Publish It

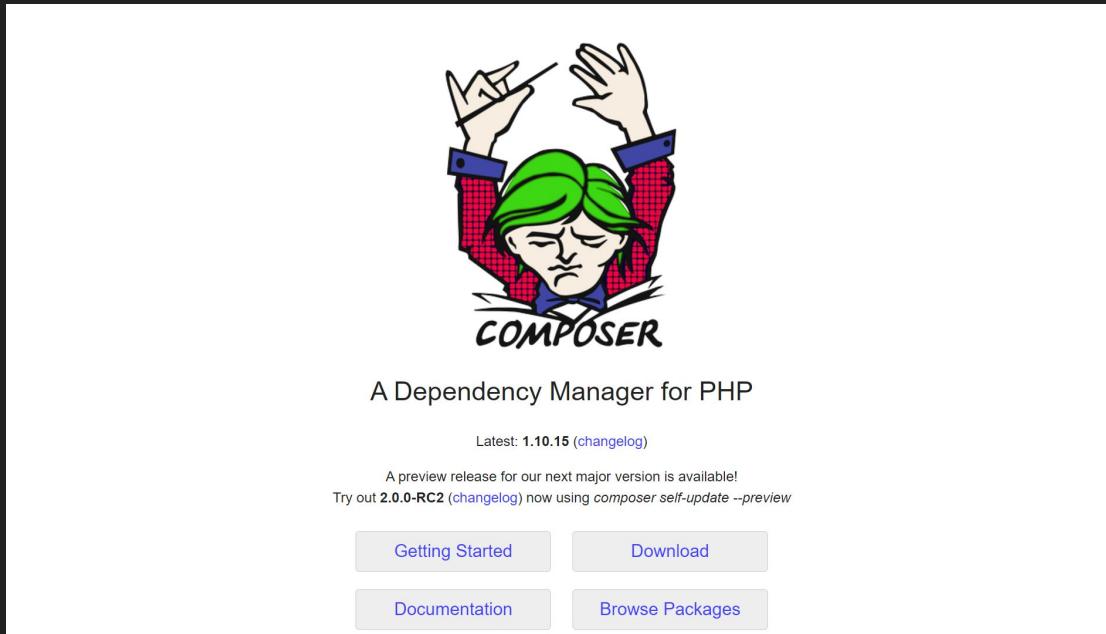
Login or register on this site, then hit the **submit** button in the menu.

Once you entered your public repository URL in there, your package will be automatically crawled periodically. You just have to make sure you keep the

# Componentes PHP

## Instalando componentes PHP

- Composer (<https://getcomposer.org/>)





## A Dependency Manager for PHP

Latest: **1.10.15** ([changelog](#))

A preview release for our next major version is available!

Try out **2.0.0-RC2** ([changelog](#)) now using *composer self-update --preview*

[Getting Started](#)

[Download](#)

[Documentation](#)

[Browse Packages](#)

[Issues](#)

[GitHub](#)

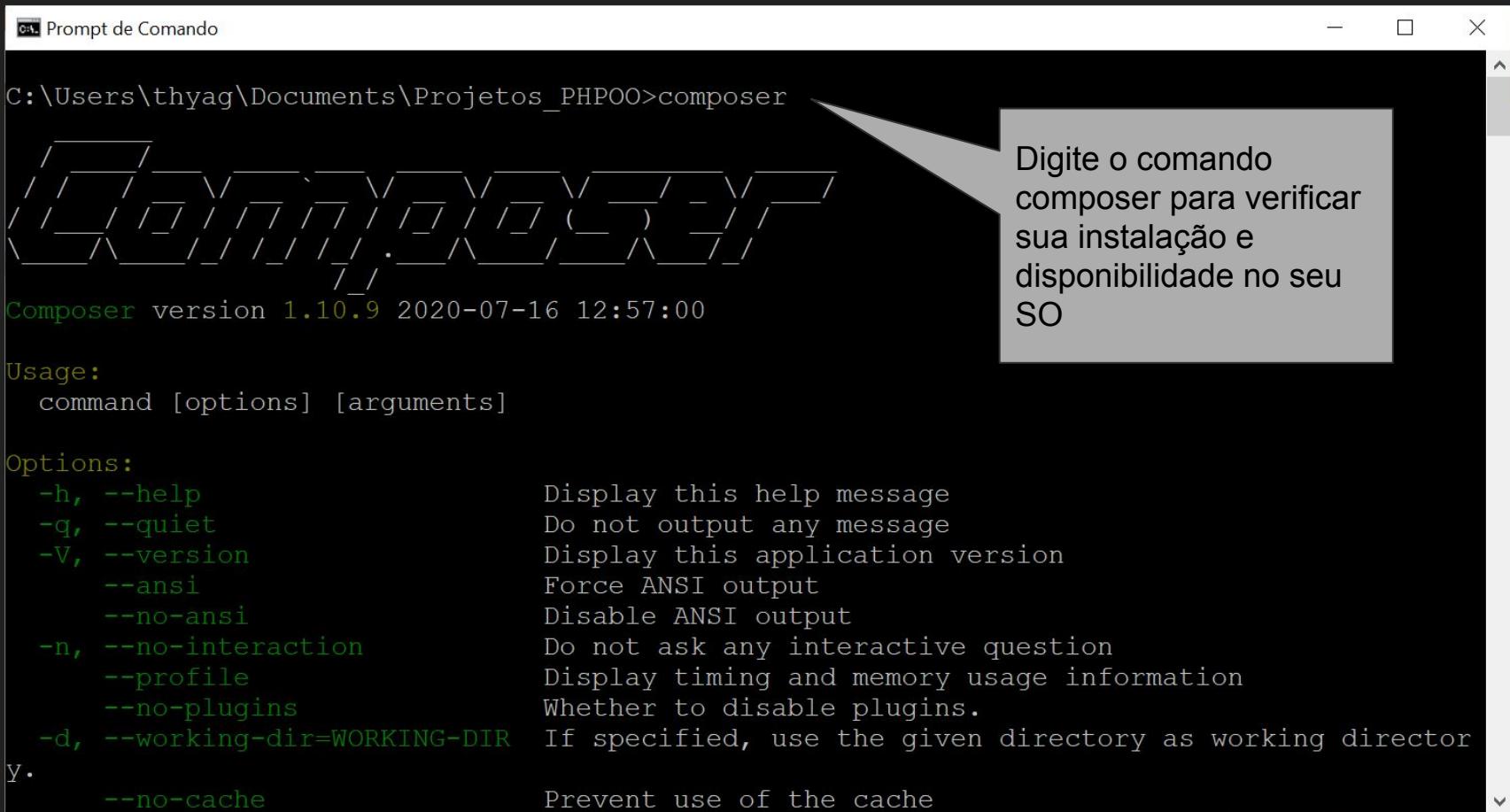


Authors: [Nils Adermann](#), [Jordi Boggiano](#) and many [community contributions](#)

Sponsored by:



Logo by: [WizardCat](#)



# Como usar o Composer



validador-cpf-cnpj

Packagist is the main Composer repository. It aggregates public PHP packages installable with Composer.

**bissolli/validador-cpf-cnpj**

Classe em PHP para validação de CPF e CNPJ.

**diego-brocane/**

Validators CPF, CNPJ

**validators-zf**

Validators for Zend Framework

PHP

13 528

9

Package type

library

2

Tags

other...

1 - Visite o Packagist e busque por um componente específico (Ex.: `validador-cpf-cnpj`)

1

## Getting Started

### Define Your Dependencies

Put a file named `composer.json` at the root of your project, containing your project dependencies:

## Publishing Packages

### Define Your Package

Put a file named `composer.json` at the root of your package's repository, containing this information:

# bissolli/validador-cpf-cnpj

↓ `composer require bissolli/validador-cpf-cnpj`

*Classe em PHP para validação de CPF e CNPJ.*

Maintainers



2 - Acesse o componente desejado, copiando o comando Composer para baixá-lo no seu projeto

[bissolli/validador-cpf-cnpj](#)

13 568

1

0

0

Stars:

13 568

Watchers:

1

Forks:

0

Open Issues:

2

1.2.2

2020-02-01 18:38 UTC

requires

requires (dev)

suggests

dev-master

1.2.2

1.2.1

```
Prompt de Comando
Microsoft Windows [versão 10.0.18363.1139]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Users\thyag>cd Documents

C:\Users\thyag\Documents>dir
O volume na unidade C é OS
O Número de Série do Volume é DE5E-15AE

Pasta de C:\Users\thyag\Documents

15/10/2020 07:56 <DIR> .
15/10/2020 07:56 <DIR> ..
26/08/2020 09:35 <DIR> Projetos_Flutter
21/10/2020 10:08 <DIR> Projetos_PHP00
16/09/2020 18:37 <DIR> Projetos_Unity
29/07/2020 08:46 <DIR> workspace-spring-tool-suite-4-4.7.0.RELEASE
    0 arquivo(s)          0 bytes
    6 pasta(s)  39.562.461.184 bytes disponíveis
```

```
C:\Users\thyag\Documents>cd Projetos_PHP00
```

```
C:\Users\thyag\Documents\Projetos_PHP00>
```

3 - Via prompt de comando ou terminal, acesse o diretório raiz do seu projeto PHP

```
C:\Users\thyag\Documents\Projetos_PHPOO>composer require bissolli/validador-cpf-cnpj
```

4 - Considerando que o Composer já esteja instalado, cole o comando Composer obtido no site Packagist e o execute.

```
C:\Users\thyag\Documents\Projetos_PHOO>composer require bissolli/validador-cpf-cnpj
Warning from https://repo.packagist.org: You are using an outdated version of Composer. Composer 2.0 is about to be released and the older 1.x releases will self-update directly to it once it is released. To avoid surprises update now to the latest 1.x version which will prompt you before self-updating to 2.x.
Using version ^1.2 for bissolli/validador-cpf-cnpj
./composer.json has been updated
Loading composer repositories with package information
Warning from https://repo.packagist.org: You are using an outdated version of Composer. Composer 2.0 is about to be released and the older 1.x releases will self-update directly to it once it is released. To avoid surprises update now to the latest 1.x version which will prompt you before self-updating to 2.x.
Updating dependencies (including require-dev)
Nothing to install or update
Generating autoload files

C:\Users\thyag\Documents\Projetos_PHOO>
```

5 - Aguarde a instalação do componente

Classe em PHP para validação de CPF e CNPJ.

## Instalação

Via Composer

```
composer require bissolli/validador-cpf-cnpj
```

## Como utilizar

Exemplo de uso para validação e formatação de CPF:

```
// Não importa se já vem formatado ou não
$document = new \Bissolli\ValidadorCpfCnpj\CPF('123.456.789.00');

// Verifica se é um número válido de CPF
// Retorna true/false
$document->isValid();

// Retorna o número de CPF formatado (###.###.###-##)
// ou false caso não seja um número válido
$document->format();

// Retorna o número de sem formatação alguma
// ou false caso não seja um número válido
$document->getValue();
```

Exemplo de uso para validação e formatação de CNPJ:

```
// Não importa se já vem formatado ou não
$document = new \Bissolli\ValidadorCpfCnpj\CNPJ('12.345.678/0001-90');

// Verifica se é um número válido de CNPJ
// Retorna true/false
```

6 - Na página do componente no Packagist, verificamos como utilizar os recursos do componente

```
 teste.php
1  <?php
2
3  require 'vendor/autoload.php';
4
5  // Não importa se já vem formatado ou não
6  $document = new \Bissolli\ValidadorCpfCnpj\CPF('123.456.789.00');
7
8  // Verifica se é um número válido de CPF
9  // Retorna true/false
10 if($document->isValid()) {
11     echo "CPF válido";
12 }
13 else {
14     echo "CPF inválido";
15 }
16
17 ?>
```

7 - Implemente um script no diretório raiz onde foi instalado o componente.

OBS.: O Composer cria e fornece um script de autoload para classes pertencentes aos componentes instalados. Logo, é necessária a inclusão do comando **require 'vendor/autoload.php'**; nos scripts que irão utilizar os componentes instalados via Composer.



```
Conta.php           ContaCorrente.php      ContaPoupanca.php      index.php          autoload.php polimorfismo    autoload.php contrato    teste.php x
?php
require 'vendor/autoload.php';

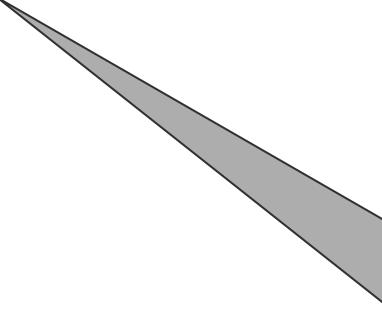
use Bissolli\ValidadorCpfCnpj\CPF;

// Não importa se já vem formatado ou não
$document = new CPF('123.456.789.00');

// Verifica se é um número válido de CPF
// Retorna true/false
if($document->isValid()) {
    echo "CPF válido";
} else {
    echo "CPF inválido";
}
```

7 - Outra forma para codificar a mesma solução

CPF inválido



8 - Teste a aplicação!

# Exercício

# Exercício



Crie uma aplicação Web que gera QR Codes de texto a partir de um componente PHP disponível no Packagist, instalando-o via Composer;

Crie um documento no Word. Nele, coloque a URL do componente escolhido, o código-fonte do script gerador e um QR Code gerado como exemplo.

Sugestão de componente: **bacon/bacon-qr-code**

# DESENVOLVIMENTO DE SISTEMAS WEB COM O FRAMEWORK LARAVEL

Thyago Maia Tavares de Farias



# O QUE É UM FRAMEWORK?



# FRAMEWORK

- Possui várias definições:
  - “Um conjunto de classe que encapsula uma abstração de projeto para a solução de uma família de problemas relacionados”;
  - “Um conjunto de objetos que colaboram para realizar um conjunto de responsabilidades para um domínio de subsistema de aplicativos”;
  - “Define um conjunto de classes abstratas e a forma como os objetos dessas classes colaboram”;
  - “Um conjunto extensível de classes orientadas a objetos que são integradas para executar conjuntos bem definidos de comportamento computacional”;
  - “Uma coleção abstraída de classes, interfaces e padrões dedicados a resolver uma classe de problemas através de uma arquitetura flexível e extensível”;



# FRAMEWORK

- Observe que um framework é uma aplicação “quase” completa, mas com “pedaços” faltando:
  - Ao receber um framework, seu trabalho consiste em prover os “pedaços” que são específicos para sua aplicação;
  - Um framework provê funcionalidades genéricas, mas pode atingir funcionalidades específicas, por configuração, durante a programação de uma aplicação;



# FRAMEWORK

- **Vantagens:**

- Maior facilidade para detecção de erros;
- Eficiência na resolução de problemas;
- Otimização de recursos;
- Concentração na abstração de solução de problemas;
- Modelados com vários padrões de projeto;



# O FRAMEWORK LARAVEL



# LARAVEL



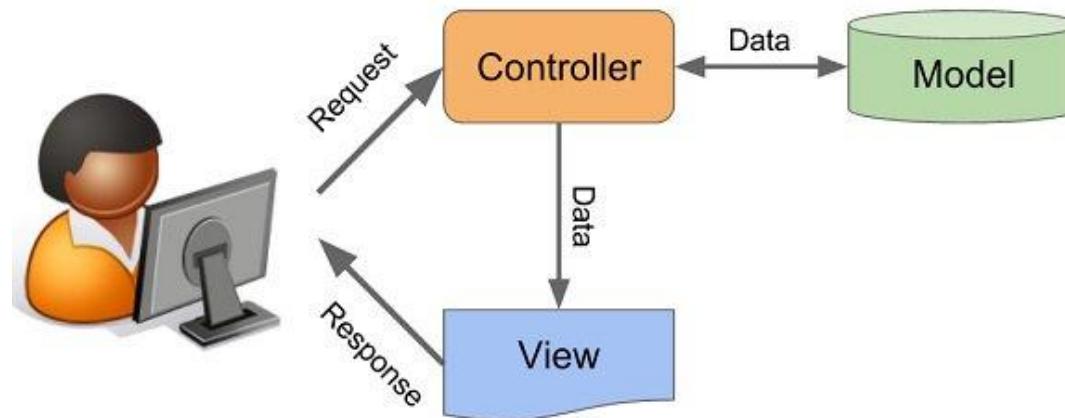
- Framework PHP open-source criado por Taylor Otwell;
- Objetiva o auxílio no desenvolvimento de aplicações Web baseados no padrão de projeto arquitetural MVC;
- Se tornou recentemente um dos frameworks PHP mais populares, ao lado do Symfony, Zend, CodeIgniter, entre outros;
- Hospedado no GitHub e licenciado nos termos da licença MIT;



# LARAVEL



- O modelo MVC:



# O AMBIENTE DE DESENVOLVIMENTO LARAVEL



# AMBIENTE DE DESENVOLVIMENTO LARAVEL

- É possível configurar um ambiente de desenvolvimento Laravel a partir do Xampp. Para isso, será necessário a instalação do **Composer**;



# AMBIENTE DE DESENVOLVIMENTO LARAVEL



- A partir de um prompt de comando, acesse a pasta **C:\xampp\htdocs** e execute o seguinte comando:

```
composer create-project laravel/laravel laravel "5.1.*"
```

- Será criada a pasta laravel na pasta htdocs do xampp, já com os arquivos de projeto Laravel. Para fazer com que o projeto fique disponível, no prompt de comando, digite:

```
cd laravel
```

```
php artisan serve
```



# AMBIENTE DE DESENVOLVIMENTO LARAVEL

- Inicie o PHP e o MySQL no control panel do Xampp, clicando nos botões Start;
- Acesse a URL **laravel.dev** (a partir do Firefox) e verifique se a página de boas vindas do Laravel será apresentada!



Laravel 5



# AMBIENTE DE DESENVOLVIMENTO LARAVEL

- Inicie o PHP e o MySQL no control panel do Xampp, clicando nos botões Start;
- Acesse a URL **localhost:8000** e verifique se a página de boas vindas do Laravel será apresentada!



Laravel 5



**REST**



# LARAVEL



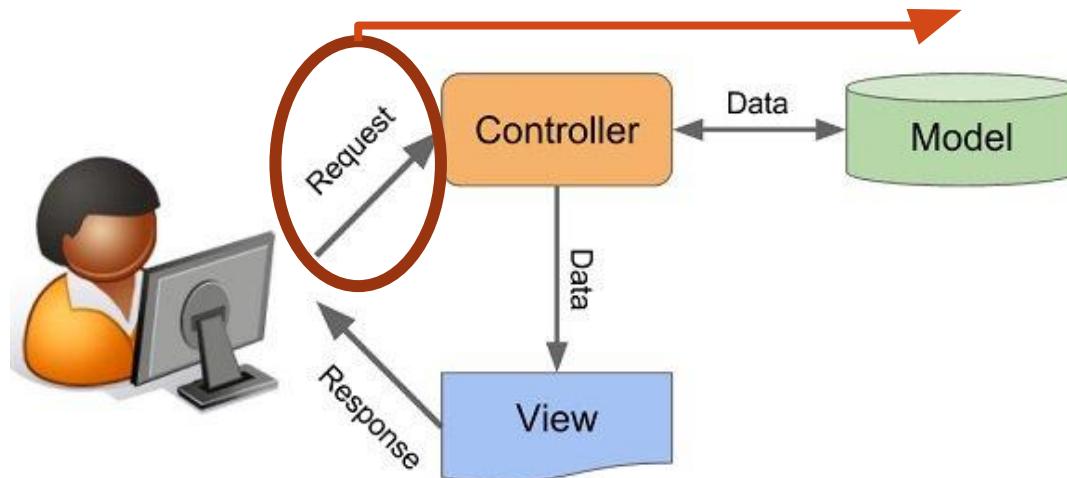
- Seleciona controllers e/ou métodos de controller a partir do modelo **REST**:
  - Cada mensagem HTTP contém toda a informação necessária para compreensão de pedidos;
  - Utiliza as **operações HTTP** para a seleção de controllers e recursos de controller: **POST, GET, PUT e DELETE**;
  - Classifica **operações de CRUD** para a persistência de dados.
    - Ex.: Quando uma requisição HTTP do tipo **DELETE** é lançada para um controller, um método de **exclusão de dados em banco** poderá ser automaticamente executada;



# LARAVEL

- O modelo MVC:

Utiliza um arquivo php de rotas para checar a operação HTTP requisitada e associá-la a um controller e/ou recurso de controller.



# ROTAS LARAVEL

O arquivo **routes/web.php**



# ROTAS LARAVEL



- A partir do arquivo **web.php**, é possível atribuir uma operação HTTP a um controller e/ou recurso de controller;
  - Ex.: Quando uma requisição HTTP do tipo get for lançada, execute o método da classe controller PrincipalController que apresenta a página de boas-vindas de uma dada aplicação web;
- Graças ao arquivo de rotas, não é necessário indicar na URL o recurso a ser acessado;
  - Ex.: Ao invés do link <http://localhost:8000/pagina.php>, poderíamos configurar uma rota para que o recurso fique acessível a partir do link <http://localhost:8000/pagina>
  - Assim, cada recurso da aplicação pode ser representado por um **verbo**;



# ROTAS LARAVEL



- Atribuições de operações HTTP são realizadas a partir da classe **Route** e de seus **métodos estáticos**, que representam cada uma das operações HTTP existentes;
- Sintaxe básica para a definição de rota:

```
Route::<operação_HTTP>('/verbo', function() {  
    // O que será feito quando essa rota for acessada  
});
```



# ROTAS LARAVEL



- Exemplo: Abra o arquivo `routes/web.php` e crie a seguinte rota:

```
Route::get('/php-info', function() {  
    phpinfo();  
});
```

- Abra o navegador, acesse <http://localhost:8000/php-info> e verifique se a página de informações sobre o servidor PHP será apresentada;



# ROTAS LARAVEL



- Exemplo 2: Abra o arquivo `routes/web.php` e crie uma nova rota:

```
Route::get('/formulario', function() {
    return '
<form method="post" action="/contato">
    Nome: <input type="text" name="nome">
    Email: <input type="text" name="email">
    Mensagem: <textarea name="mensagem"></textarea>
    <input type="submit" value="Enviar">
</form>';
});
```



# ROTAS LARAVEL

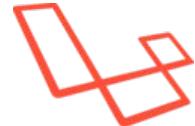


- Exemplo 2: Abra o arquivo `routes/web.php` e crie uma nova rota:

```
Route::post('/contato', function() {  
    print_r(Request::all());  
});
```



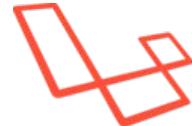
# ROTAS LARAVEL



- Abra o navegador, acesse <http://localhost:8000/formulario> e verifique se o formulário criado no arquivo de rotas será apresentado. Digite suas informações de contato e os submeta, afim de verificar se a rota post será executada;
- Algum problema?



# ROTAS LARAVEL



- Abra o navegador, acesse <http://localhost:8000/formulario> e verifique se o formulário criado no arquivo de rotas será apresentado. Digite suas informações de contato e os submeta, afim de verificar se a rota post será executada;
- Algum problema? **SIM**
  - Todo formulário Laravel precisa submeter um **Token**, chamado **CSRF**, para que possa enviar operações HTTP em aplicações Laravel (por questões de segurança!);



# ROTAS LARAVEL



- Como só podemos inserir tais tokens em formulários implementados em **Views** (olha o MVC aí de novo!), por enquanto, vamos desligar esse recurso, comentando a linha de código que implementa esse recurso no arquivo **app/Http/Kernel.php**;

```
/**
 * The application's global HTTP middleware stack.
 *
 * @var array
 */
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    //\App\Http\Middleware\VerifyCsrfToken::class,
];
```



# ROTAS LARAVEL



- Abra o navegador, acesse <http://localhost:8000/formulario> e verifique se o formulário criado no arquivo de rotas será apresentado. Digite suas informações de contato e os submeta, afim de verificar se a rota post será executada;



# ROTAS LARAVEL



- Métodos da classe `Route` disponíveis para rotas:
- `Route::get($uri, $callback);` - Utilizado de forma geral para navegar entre páginas;
- `Route::post($uri, $callback);` - Utilizado de forma geral para alterações no lado do servidor;
- `Route::put($uri, $callback);` - Utilizado de forma geral para atualizações de um recurso existente;
- `Route::patch($uri, $callback);` - Utilizado de forma geral para atualizações de um recurso existente;
- `Route::delete($uri, $callback);` - Utilizado de forma geral para remover um recurso existente;



# ROTAS LARAVEL



- Também é possível definir vários verbos para uma única rota com o método **match**:

```
Route::match(['POST', 'GET'], '/formulario', function() {
    if(Request::isMethod('post'))
        print_r(Request::all());
    else {
        return '
<form method="post" action="/formulario">
Nome: <input type="text" name="nome">
Email: <input type="text" name="email">
Mensagem: <textarea name="mensagem"></textarea>
<input type="submit" value="Enviar">
</form>';
    }
});
```



# ROTAS LARAVEL



- Também é possível definir **todos os verbos** para uma única rota com o método **any**:

```
Route::any('/formulario', function() {
    if(Request::isMethod('post'))
        print_r(Request::all());
    else {
        return '
<form method="post" action="/formulario">
Nome: <input type="text" name="nome">
Email: <input type="text" name="email">
Mensagem: <textarea name="mensagem"></textarea>
<input type="submit" value="Enviar">
</form>';
    }
});
```



# ROTAS COM PARÂMETROS

O arquivo `routes/web.php`



# ROTAS LARAVEL



- Até o momento, não passamos nenhum parâmetro nas nossas requisições (rotas). Mas em muitas aplicações, essa necessidade pode surgir;
- Para isso, a sintaxe na definição do verbo muda um pouco, onde será necessário explicitar os parâmetros na sua definição;



# ROTAS LARAVEL



- Exemplo: Abra o arquivo `routes/web.php` e crie a seguinte rota:

```
Route::get('/contato/{id}', function($id) {  
    printf('Olá, o seu ID é %s', $id);  
});
```

- Abra o navegador, acesse <http://localhost:8000/contato/25> e verifique se o ID 25 é apresentado no navegador. Se nenhum parâmetro for enviado, será gerada uma exceção;



# ROTAS LARAVEL



- Já que nossa rota espera um id, vamos alterá-la para aceitar apenas números (atualmente ela também aceita strings!);
- Para isso, basta invocar o método **where** após a definição da rota, passando como parâmetro o nome do parâmetro e a expressão regular que impõe o uso de números;



# ROTAS LARAVEL



- Exemplo: Abra o arquivo `routes/web.php` e crie a seguinte rota:

```
Route::get('/contato/{id}', function($id) {  
    printf('Olá, o seu ID é %s', $id);  
})->where('id', '[0-9]+');
```

- Abra o navegador, acesse <http://localhost:8000/contato/sport> e verifique se o ID 'Sport' é apresentado no navegador. Se nenhum parâmetro for enviado, será gerada uma exceção;



# ROTAS LARAVEL



- Exemplo: Podemos passar mais de um parâmetro. Abra o arquivo **routes/web.php** e edite a seguinte rota:

```
Route::get('/contato/{id}/{nome}', function($id, $nome)
{
    printf('Olá %s, o seu ID é %s', $nome, $id);
});
```

- Abra o navegador, acesse <http://localhost:8000/contato/25/Thyago> e verifique a saída apresentada no navegador. Se nenhum parâmetro for enviado, será gerada uma exceção;



# ROTAS E CONTROLADORES



# ROTAS LARAVEL



- E se minha aplicação tiver mil rotas????
- Você não precisa criar todas as rotas no arquivo web.php!
- Podemos **mapear um controller no arquivo de rotas** e, a partir dele, aplicar as rotas;
- Para isso, em cada parâmetro passado na rota, devemos criar o parâmetro correspondente no método controller;



# ROTAS LARAVEL



- O melhor? O Laravel cria o controller para você automaticamente!!!
- Para isso, vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\nomeProjeto** e digite o comando:

```
php artisan make:controller Rotas
```

- Um novo controller com o nome Rotas será criado em **app/Http/Controllers**;



# ROTAS LARAVEL



- Agora, só precisaremos adicionar duas linhas no arquivo de rotas:

```
use App\Http\Controllers\Rotas;  
Route::get('/rotas', [Rotas::class, 'index']);
```

- Abra o navegador, acesse <http://localhost:8000/rotas>. Como ainda não existe nenhum método de rota, será gerada uma exceção;
- Vamos atualizar nosso controller para que tenha um método padrão, a ser executado quando nenhuma rota for especificada;



# ROTAS LARAVEL



- No controller `app/Http/Controllers/Rotas.php`, adicione o seguinte método:

```
public function index()      {
    return 'Olá, sou a rota padrão do controller!';
}
```

- Abra o navegador, acesse <http://localhost:8000/rotas>



# ROTAS LARAVEL



- Também podemos criar o nome que desejarmos para nossas rotas. Exemplo: Logo abaixo do método getIndex, crie o método:

```
public function getRequisicao()      {  
    return 'Olá, sou executado a partir de um GET';  
}
```

- No arquivo de rotas, adicione a seguinte rota:

```
Route::get('/teste', 'Rotas@getRequisicao');
```

- Abra o navegador, acesse <http://localhost:8000/teste>



# MIDDLEWARE HTTP



# MIDDLEWARE HTTP



- Classes Laravel utilizadas para filtrar os dados de entrada dos nossos sistemas;
- Funciona como um filtro de requisição, que executa ações antes ou depois de uma requisição HTTP;



# MIDDLEWARE HTTP



- Exemplo: Podemos inserir um token CSRF em formulários implementados em **Views** para que apenas nossos forms realizem requisições. Precisamos religar esse recurso, retirando o comentário da linha de código que implementa esse recurso no arquivo **app/Http/Kernel.php**:

```
/**
 * The application's global HTTP middleware stack.
 *
 * @var array
 */
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    //\App\Http\Middleware\VerifyCsrfToken::class,
];
```



# MIDDLEWARE HTTP



- Exemplo: Agora, vamos criar nossa primeira View! Vá para a pasta `resources/views` e crie o arquivo `contato.blade.php`. Edite-o com o seguinte código HTML:

```
<html>
  <body>
    <form method="post" action="contato">
      {{ csrf_field() }}
      <p>Nome: <input type="text" name="nome"></p>
      <p>Telefone: <input type="text" name="telefone"></p>
      <p><input type="submit" value="Enviar"></p>
    </form>
  </body>
</html>
```



# MIDDLEWARE HTTP



- Exemplo: Agora, vamos criar nossa primeira View! Vá para a pasta `resources/views` e crie o arquivo `contato.blade.php`. Edite-o com o seguinte código HTML:

```
<html>
  <body>
    <form method="post" action="contato">
      {{ csrf_field() }} ← Olha o middleware aqui gente!!!
      <p>Nome: <input type="text" name="nome"></p>
      <p>Telefone: <input type="text" name="telefone"></p>
      <p><input type="submit" value="enviar"></p>
    </form>
  </body>
</html>
```



# MIDDLEWARE HTTP



- Exemplo: Agora, vamos criar uma rota para chamar nosso formulário de contato no arquivo de rotas:

```
use App\Http\Controllers\ContatoController;

Route::get('/contato', [ContatoController::class,
    'contato']);

Route::post('/contato', [ContatoController::class,
    'enviarContato']);
```



# MIDDLEWARE HTTP



- Em seguida, criaremos o controller responsável por exibir e receber dados do formulário;
- Para isso, vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\nomeProjeto** e digite o comando:

```
php artisan make:controller ContatoController
```

- Um novo controller com o nome **ContatoController** será criado em **app/Http/Controllers**;



# MIDDLEWARE HTTP



- Precisaremos criar os métodos `contato()` e `enviarContato(Request $request)` no controller em questão:

```
public function contato()      {
    return view('contato');
}

public function enviarContato(Request $request)      {
    dd($request->all());
    // dd: dump and die - Função Laravel para dump em variáveis
}
```

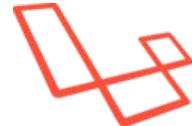
- Abra o navegador, acesse <http://localhost:8000/contato>



# CONTROLADORAS (CONTROLLERS)



# CONTROLLERS



- Até o momento, criamos uma controladora que responde a requisições HTTP do tipo GET e POST;
- Mas... E quando eu precisar responder outros tipos de requisição, como PUT ou DELETE?;



# CONTROLLERS



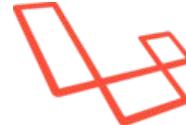
- Exemplo: Vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\nomeProjeto** e digite o comando:

```
php artisan make:controller BasicoController
```

- Um novo controller com o nome BasicoController será criado em **app/Http/Controllers**;



# MIDDLEWARE CUSTOMIZADOS

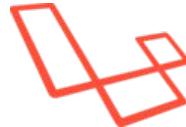


- No controller **BasicoController**, adicione os seguintes métodos:

```
public function putBasico()      {  
    return view('put');  
}  
  
public function put (Request $request)      {  
    echo ("Nome digitado: " . $request->input('nome'));  
}
```



# CONTROLLERS



- Em seguida, vamos atualizar nosso arquivo de rotas:

```
use App\Http\Controllers\BasicoController;

Route::get('/put', [BasicoController::class,
'putBasico']);

Route::put('/put', [BasicoController::class, 'put']);
```



# CONTROLLERS



- Agora, começaremos a manipular a camada de visualização de dados. Crie a view `put.blade.php` na pasta `resource/views`:

```
<html>
<body>
    <form method="post" action="/put">
        {{ csrf_field() }}
        <input type="hidden" name="_method" value="PUT">
        <p>Nome: <input required type="text" name="nome"></p>
    </form>
</body>
</html>
```



# CONTROLLERS



- Abra o navegador e acesse <http://localhost:8000/put>



# CONTROLLERS



- Uma outra forma de alterar o tipo de requisição em um form HTML:

```
<html>
<body>
    <form method="post" action="/put">
        {{ csrf_field() }}
        {{ method_field('PUT') }}
        <p>Nome: <input required type="text" name="nome"></p>
    </form>
</body>
</html>
```



# CONTROLLERS DE RECURSO



# CONTROLLERS DE RECURSO

- Controllers também podem ser concebidas com o objetivo de abstrair funcionalidades CRUD (Criação, Leitura, Atualização e Exclusão);
- O Laravel pode não só criar tais tipos de classe automaticamente, como também as assinaturas de método necessárias e associar um tipo de requisição para cada método;



# CONTROLLERS DE RECURSO

- Exemplo: Vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\laravel** e digite o comando:

```
php artisan make:controller LivroController --resource
```

- Um novo controller com o nome LivroController será criado em **app/Http/Controllers**;



# CONTROLLERS DE RECURSO

- Verifique que o Laravel já criou as assinaturas de método de CRUD automaticamente:
  - `index()`
  - `create()`
  - `store(Request $request)`
  - `show($id)`
  - `edit($id)`
  - `update(Request $request, $id)`
  - `destroy($id)`



# CONTROLLERS DE RECURSO

- Outro poder de controladoras de recurso é a simplificação do registro de rotas. Continuando nosso exemplo, acesse o arquivo de rotas e registre a seguinte rota:

```
use App\Http\Controllers\LivroController;  
  
Route::resource('livro', LivroController::class);
```



# CONTROLLERS DE RECURSO

- Com esse registro de rota, o Laravel automaticamente atribui as seguintes requisições HTTP e as rotas para cada método da controladora de recurso:

| Tipo      | URI              | Ação    |
|-----------|------------------|---------|
| GET       | /livro           | index   |
| GET       | /livro/create    | create  |
| POST      | /livro           | store   |
| GET       | /livro/{id}      | show    |
| GET       | /livro/{id}/edit | edit    |
| PUT/PATCH | /livro/{id}      | update  |
| DELETE    | /livro/{id}      | destroy |



# CONTROLLERS DE RECURSO

- Continuando o exemplo, no controller **LivroController**, edite os seguintes métodos:

```
public function index()      {  
    return 'Olá, usuário!';  
}  
public function create ()    {  
    return 'Aqui acessarei views para a inserção de dados!';  
}
```



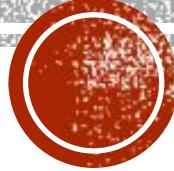
# CONTROLLERS DE RECURSO

- Abra o navegador e acesse <http://localhost:8000/livro> e <http://localhost:8000/livro/create> para testar as rotas baseadas em métodos do nosso controller de recurso;



# RECURSOS DE LAYOUT

Recursos dinâmicos fornecidos pelo Blade



# RECURSOS DE LAYOUT



- É possível utilizar PHP puro nas views para geração de conteúdo dinâmico. Mas isso é uma boa prática???
- O ideal é só inserir scripts PHP nos controllers e models!
- O **Blade** fornece uma sintaxe e instruções que tornam a codificação de Views mais enxuta e organizada;



# RECURSOS DE LAYOUT



- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/exibirdados', function() {  
    return view('exibirdados');  
});
```



# RECURSOS DE LAYOUT



- Crie na pasta `/resources/views` a view `exibirdados.blade.php` e acesse <http://localhost:8000/exibirdados> :

```
<h1>{{ $titulo ?? 'Título não encontrado' }}</h1>
<p>{{ $texto ?? 'Texto não encontrado' }}</p>
```

- If/else muito enxuto!



# RECURSOS DE LAYOUT



- Exemplo: Agora, acesse o arquivo de rotas , atualize a rota **exibirdados** e acesse <http://localhost:8000/exibirdados> :

```
Route::get('/exibirdados', function() {
    return view('exibirdados', [
        'titulo' => 'Meu Blog',
        'texto'  => 'Sejam bem vindos!'
    ]);
});
```



# RECURSOS DE LAYOUT



- O Blade também permite **escapar dados** de maneira simples;
- Escapar dados é muito importante por questões de segurança. Assim, evitamos que o usuário envie dados maliciosos a partir de técnicas como **SQL injection** e **XSS**;



# RECURSOS DE LAYOUT



- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/naoesc', function() {  
    return view('naoesc', [  
        'conteudo' => '<h1>Sport</h1>'    ]);  
}) ;
```



# RECURSOS DE LAYOUT



- Crie na pasta `/resources/views` a view `naoesc.blade.php` e acesse <http://localhost:8000/naoesc> :

```
{{ $conteudo }}
```

- Sem escape, as tags HTML que foram enviadas como parâmetros de view foram enviadas para a aplicação (eu poderia ter enviado um Javascript malicioso, concorda?);



# RECURSOS DE LAYOUT



- Agora edite a view `naoesc.blade.php` e acesse <http://laravel.dev/naoesc> :

```
{!! $conteudo !!}
```

- Com `escape`, as tags HTML que foram enviadas como parâmetros de view foram “escapadas”;



# RECURSOS DE LAYOUT



- O Blade também permite **inserir estruturas de controle e repetição** em views;
- Tais estruturas auxiliam na geração de conteúdo dinâmico a partir de uma sintaxe bastante enxuta;



# RECURSOS DE LAYOUT



- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/se', function() {  
    return view('vazio', [  
        'lista' => []  
    ]);  
});
```



# RECURSOS DE LAYOUT



- Agora edite a view **vazio.blade.php** e acesse <http://localhost:8000/se> :

```
@if(count($lista) == 0)
    <p>Não existe dados na lista</p>
@else
    <p>Existem dados na lista</p>
@endif
```



# RECURSOS DE LAYOUT



- Acesse o arquivo de rotas, atualize a rota se e acesse <http://localhost:8000/se> :

```
Route::get('/se', function() {  
    return view('vazio', [  
        'lista' => ['item']  
    ]);  
});
```



# RECURSOS DE LAYOUT



- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/para-cada', function() {  
    return view('dados', [  
        'lista' => ['dado1', 'dado2']  
    ]);  
});
```



# RECURSOS DE LAYOUT



- Agora edite a view `dados.blade.php` e acesse <http://localhost:8000/para-cada> :

```
@foreach($lista as $item)  
    <p>{{ $item }}</p>  
@endforeach
```



# RECURSOS DE LAYOUT



- Para permitir o reuso de código, geralmente dividimos nossas Views em pequenas partes e, depois, as utilizamos quando necessário;
- Assim, é possível alterar partes de View em um só lugar e refletir a alteração em vários lugares ao mesmo tempo. O Blade nos ajuda nesse sentido;



# RECURSOS DE LAYOUT



- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/include', function() {  
    return view('include');  
});
```



# RECURSOS DE LAYOUT



- Crie na pasta `/resources/views` a view `include.blade.php` e acesse <http://localhost:8000/include> :

```
<h1>Parte da View principal</h1>
@include('welcome')
```



# RECURSOS DE LAYOUT



- O Blade também fornece estruturas de repetição.  
Ex:

```
@for ($i=0;$i<10;$i++)  
    <p>O valor de i é {{ $i }}</p>  
@endfor
```



# RECURSOS DE LAYOUT



- O Blade também fornece estruturas de repetição.  
Ex:

```
@while(true)  
  <p>Vou travar seu navegador!</p>  
@endwhile
```



# VALIDAÇÕES DE DADOS



# VALIDAÇÕES DE DADOS



- O Laravel nos propõe facilidades para a validação de dados a partir de controllers;
- Para apresentar o referido recurso, vamos implementar, na forma tradicional, um controller com métodos de validação. Logo em seguida, iremos refatorar a classe com recursos de validação do Laravel;



# VALIDAÇÕES DE DADOS



- Exemplo: Vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\laravel** e digite o comando:

```
php artisan make:controller ValidacaoController
```

- Um novo controller com o nome **ValidacaoController** será criado em **app/Http/Controllers**;



# VALIDAÇÕES DE DADOS



- Continuando o exemplo, no controller `ValidacaoController`, crie o seguinte método:

```
public function validaTitulo(Request $request)    {  
    if($request->input('titulo') == '')  
        return 'Título não pode ser vazio';  
    if(strlen($request->input('titulo')) < 3)  
        return 'Título deve ter no mínimo 3 letras';  
}
```



# VALIDAÇÕES DE DADOS



- Agora, no controller `ValidacaoController`, iremos refatorar o primeiro if do método `validaTitulo`:

```
public function validaTitulo(Request $request)      {
    $this->validate($request, [
        'titulo' => 'required'
    ]);
    if(strlen($request->input('titulo')) < 3)
        return 'Título deve ter no mínimo 3 letras';
}
```



# VALIDAÇÕES DE DADOS



- Em seguida, no controller **ValidacaoController**, iremos refatorar a classe, tirando a necessidade do segundo if do método **validaTitulo**:

```
public function validaTitulo(Request $request)      {  
    $this->validate($request, [  
        'titulo' => 'required|min:3'  
    ]);  
}
```



# VALIDAÇÕES DE DADOS



- Para testar nossas validações, acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/form', function() {  
    return view('form');  
});  
Route::post('/form',  
    'ValidacaoController@validaTitulo' );
```



# VALIDAÇÕES DE DADOS



- Agora, começaremos a manipular a camada de visualização de dados. Crie a view `form.blade.php` na pasta `resources/views` e acesse <http://localhost:8000/form> :

```
@if(count($errors) > 0)
    @foreach($errors->all() as $error)
        <p>{{ $error }}</p>
    @endforeach
@endif

<form method="post">
    {{ csrf_field() }}
    <p>Titulo: <input type="text" name="titulo"></p>
    <input type="submit" value="Enviar">
</form>
```



# ATIVIDADE PRÁTICA



# ATIVIDADE PRÁTICA

- Faça uma aplicação no Laravel que:
  - Na página raiz da aplicação, permita ao usuário visualizar 4 links: Adição, Subtração, Multiplicação e Divisão. Cada link deverá levar o usuário a uma rota criada para cada operação;
  - Na pasta Views, crie uma View para cada operação matemática, além de uma View para a página de links;
    - Nas Views de cálculo, o usuário deverá digitar dois números em um formulário;
  - Crie um controller que possua 4 métodos, cada um dedicado para uma operação matemática;
  - Edite o arquivo de rotas para que uma rota seja definida para cada um dos métodos da classe controller criada anteriormente;



# ATIVIDADE PRÁTICA

- Observação:
  - Para criar um método que possa capturar os inputs do formulário, utilize um objeto da classe Request como parâmetro do método.  
Exemplo:

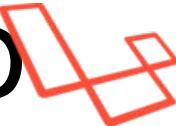
```
public function postNome(Request $request) {  
    printf("Nome: %s", ($request->input('nome')));  
}
```



# CONFIGURANDO BD NO LARAVEL



# CONFIGURANDO BD



- As configurações de banco de dados no Laravel ficam armazenadas na pasta **config**, arquivo **database.php**;
- Como exemplo, edite o vetor **connections** do arquivo **database.php** com as configurações apresentadas ao lado

```
'mysql' => [  
    'driver'      => 'mysql',  
    'host'        => env('DB_HOST', 'localhost'),  
    'database'    => env('DB_DATABASE', 'laravel'),  
    'username'    => env('DB_USERNAME', 'root'),  
    'password'    => env('DB_PASSWORD', ''),  
    'charset'     => 'utf8',  
    'collation'   => 'utf8_unicode_ci',  
    'prefix'      => '',  
    'strict'      => false,  
,
```



# CONFIGURANDO BD

- Crie a base de dados “laravel” a partir do **phpmyadmin**;
- Abra o arquivo **.env** localizado na pasta raiz do projeto Laravel e edite as configurações padrão de conexão;

```
5 DB_CONNECTION=mysql
6 DB_HOST=127.0.0.1
7 DB_DATABASE=laravel
8 DB_USERNAME=root
9 DB_PASSWORD=
```



# MIGRAÇÕES



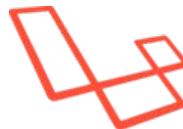
# MIGRAÇÕES



- Funcionalidade nativa do Laravel;
  - O primeiro framework PHP a se preocupar com isso;
  - Algo que era disponível apenas no Ruby!
- Permite a criação de classes que permitem inicializar (subir) bancos de dados de aplicações e remover informações de bancos quando necessário;
- Facilita a sincronização de dados em banco;



# MIGRAÇÕES



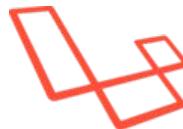
- Exemplo: Vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\nomeProjeto** e digite o comando:

```
php artisan make:migration tabela_livro --create=livro
```

- Uma nova migration com o nome **xxxx\_tabela\_livro** será criada em **database/migrations/**;



# MIGRAÇÕES

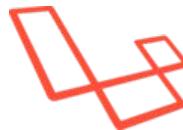


- Continuando o exemplo, na migration `tabela_livro`, edite só o método `up`:

```
public function up() {  
    Schema::create('livro', function(Blueprint $table) {  
        $table->bigIncrements('id');  
        $table->string('nome');  
        $table->timestamps();  
    });  
}
```



# MIGRAÇÕES



Vá até a pasta App/Providers e abra a classe AppServiceProvider, dentro do método boot adicione o seguinte:

```
use Illuminate\Support\Facades\Schema;

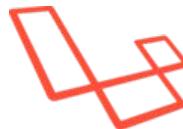
public function boot()
{
    Schema::defaultStringLength(191);
}
```

[Illuminate\Database\QueryException] SQLSTATE[42000]: Syntax error or access violation: 1071 Specified key was too long; max key length is 767 bytes (SQL: alter table `users` add unique `users_email_unique (email)`)

[PDOException] SQLSTATE[42000]: Syntax error or access violation: 1071 Specified key was too long; max key length is 767 bytes



# MIGRAÇÕES



- Exemplo: Vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\nomeProjeto** e digite o comando:

```
php artisan migrate
```

- Quando o comando for concluído, abra o **phpmyadmin**, banco **laravel** e verifique se a tabela **livros** foi criada com sucesso;



# MIGRAÇÕES



- Exemplo: Para executar o método down da migration, vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\ nomeProjeto** e digite o comando:

```
php artisan migrate:reset
```

- Quando o comando for concluído, abra o **phpmyadmin**, banco **laravel** e verifique se a tabela **livros** foi excluída com sucesso;

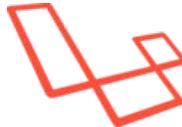


# ELOQUENT ORM

Chegou a hora de criar nossas classes Model!!



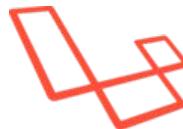
# ELOQUENT ORM



- O Laravel trabalha com **ORM** (Object Relacional Mapping), que se encarrega de abstrair a conversão das estruturas de tabelas de BD e instruções SQL para o mundo da orientação a objetos;
- O **Eloquent** implementa o padrão Active Record, em que cada tabela deve possuir uma classe correspondente (um Model MVC);



# ELOQUENT ORM



- Exemplo: Vá para a pasta **app** do projeto laravel e crie a pasta **Models** (infelizmente o Laravel não cria essa pasta por padrão);
- Vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\ nomeProjeto** e digite o comando:

```
php artisan make:model Models/LivroModel
```

- Uma nova model com o nome **LivroModel** será criada em **app/Models/**;



# ELOQUENT ORM



- Continuando o exemplo, edite a Model **LivroModel**:

```
class LivroModel extends Model {  
    protected $table = 'livro';  
    protected $primaryKey = 'id';  
    protected $fillable = ['nome'];  
}
```



# ELOQUENT ORM



- Para testar nossas consultas, acesse o arquivo de rotas e registre a seguinte rota:

```
Route::resource('livro', 'LivroController');
```



# ELOQUENT ORM



- Agora, vá no seu terminal, acesse a partir dele a pasta **C:\xampp\htdocs\nomeProjeto** e digite o comando:

```
php artisan make:controller LivroController --resource
```

- Uma nova controller com o nome **LivroController** será criada em **app/Http/Controllers/**;



# ELOQUENT ORM



- Continuando o exemplo, inclua uma importação na classe **LivroController**:

```
use App\Models\LivroModel;
```



# ELOQUENT ORM



- Em seguida, edite o método **index** da classe **LivroController**:

```
public function index() {  
    return LivroModel::all();  
    /* all(): Método do Eloquent para o select * from  
    livro */  
}
```



# ELOQUENT ORM



- Por fim, abra o **phpmyadmin**, acesse a tabela **livros**, insira alguns livros (clicando em **Inserir**), abra o navegador e acesse <http://localhost:8000/livro/>
- Um **select \* from livros** será executado automaticamente e apresentado na tela!



# ATIVIDADE PRÁTICA



# ATIVIDADE PRÁTICA

- Edite o exemplo anterior e faça com que o projeto Laravel também permita inserir, editar e excluir livros, além de exibir informações de um livro específico;
- A sintaxe para os métodos de Model para as devidas ações CRUD estão disponíveis em  
<https://laravel.com/docs/5.8/eloquent>
- Faça os registros de rota e a criação de Views necessárias para cada operação;



# ATIVIDADE PRÁTICA

- Observação:
  - Para criar um método que possa capturar os inputs do formulário, utilize um objeto da classe Request como parâmetro do método.  
Exemplo:

```
public function postNome(Request $request) {  
    printf("Nome: %s", ($request->input('nome')));  
}
```



# ATIVIDADE PRÁTICA

- Lembre-se também que cada método CRUD criado em um controller de recurso pode ser executado automaticamente, dependendo da requisição HTTP realizada e do formato da URI:

| Tipo      | URI              | Ação    |
|-----------|------------------|---------|
| GET       | /livro           | index   |
| GET       | /livro/create    | create  |
| POST      | /livro           | store   |
| GET       | /livro/{id}      | show    |
| GET       | /livro/{id}/edit | edit    |
| PUT/PATCH | /livro/{id}      | update  |
| DELETE    | /livro/{id}      | destroy |

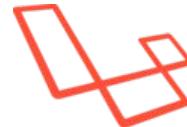


# ATIVIDADE PRÁTICA

- Lembre-se também que cada método CRUD criado em um controller de recurso tem uma funcionalidade específica sugerida:
  - **index** – Método que chamará uma view para a exibição de dados;
  - **create** – Método que chamará uma view para o formulário de inserção de dados;
  - **store** – Método que executará a inserção de dados a partir de um Model;
  - **show** - Método que chamará uma view para a exibição de um recurso específico;
  - **edit** - Método que chamará uma view para o formulário de edição de dados;
  - **update** - Método que executará a edição de dados a partir de um Model;
  - **destroy** – Método que executará a exclusão de dados a partir de um Model;



# ELOQUENT ORM

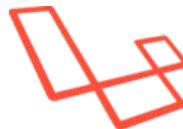


- Exemplo: Edite o método **store** da classe **LivroController**:

```
public function store(Request $request) {  
    $livro = new LivroModel;  
    $livro->nome = $request->input('nome');  
    $livro->save();  
  
    return LivroModel::all();  
}
```



# ELOQUENT ORM



- Exemplo: Edite o método **create** da classe **LivroController**:

```
public function create() {  
    return view('cadastro');  
}
```



# ELOQUENT ORM



- Exemplo: Crie a view `cadastro.blade.php` para o cadastro de livros:

```
<html>
<body>
  <form method="post" action="/livro">
    {{ csrf_field() }}
    <p>Nome: <input type="text" name="nome"></p>
  </form>
</body>
</html>
```



# ELOQUENT ORM



- Abra o navegador e acesse <http://localhost:8000/livro> e <http://localhost:8000/livro/create> para testar as rotas baseadas em métodos do nosso controller de recurso;



# ELOQUENT ORM

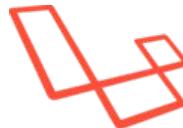


- Exemplo: Edite o método **update** da classe **LivroController**:

```
public function update(Request $request, $id) {  
    $livro = LivroModel::find($id);  
    $livro->nome = $request->input('nome');  
    $livro->save();  
  
    return LivroModel::all();  
}
```



# ELOQUENT ORM



- Exemplo: Edite o método **edit** da classe **LivroController**:

```
public function edit($id) {  
    return view('edita', [  
        'id' => $id  
    ]);  
}
```



# ELOQUENT ORM

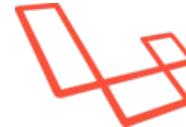


- Exemplo: Crie a view `edita.blade.php` para a edição de livros:

```
<html>
<body>
    <form method="post" action="/livro/{{ $id }}">
        {{ csrf_field() }}
        {{ method_field('PUT') }}
        <p>Nome: <input type="text" name="nome"></p>
    </form>
</body>
</html>
```



# ELOQUENT ORM



- Abra o navegador e acesse <http://localhost:8000/livro/1/edit> para testar as rotas baseadas em métodos do nosso controller de recurso;



# ELOQUENT ORM



- Exemplo: Edite o método **edit** da classe **LivroController**:

```
public function edit($id) {  
    return view('edita', [  
        'id' => $id,  
        'nome' => $this->show($id)->nome  
    ]);  
}
```



# ELOQUENT ORM



- Exemplo: Crie a view `edita.blade.php` para a edição de livros:

```
<html>
<body>
    <form method="post" action="/livro/{{ $id }}">
        {{ csrf_field() }}
        {{ method_field('PUT') }}
        <p>Nome: <input type="text" name="nome" value="{{ $nome }}"/></p>
    </form>
</body>
</html>
```



# ELOQUENT ORM



- Abra o navegador e acesse <http://localhost:8000/livro/1/edit> para testar as rotas baseadas em métodos do nosso controller de recurso;



# ELOQUENT ORM



- Exemplo: Edite o método **show** da classe **LivroController**:

```
public function show($id) {  
    return LivroModel::find($id);  
}
```



# ELOQUENT ORM



- Abra o navegador e acesse <http://localhost:8000/livro/1> para testar as rotas baseadas em métodos do nosso controller de recurso;



# ELOQUENT ORM

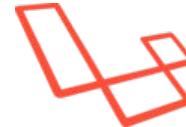


- Exemplo: Edite o método **destroy** da classe **LivroController**:

```
public function destroy($id) {  
    $livro = LivroModel::find($id);  
    $livro->delete();  
  
    return LivroModel::all();  
}
```



# ELOQUENT ORM



- Exemplo: Atuaize a view `edita.blade.php` para a exclusão de livros:

```
<html>
<body>
    ...
    <form method="post" action="/livro/{{ $id }}">
        {{ csrf_field() }}
        {{ method_field('DELETE') }}
        <p><input type="submit" value="EXCLUIR"></p>
    </form>

</body>
</html>
```



# ELOQUENT ORM



- Abra o navegador e acesse <http://localhost:8000/livro/1/edit> e clique em excluir para testar as rotas baseadas em métodos do nosso controller de recurso;

