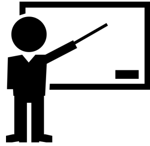
INSTITUTO FEDERAL
Paraná
Campus Paranaguá

Ministério da Educação



TECNÓLOGO EM ANÁLISE E DESENVOLVIMENTO
Disciplina de Desenvolvimento Web II
Aula 08: Framework Laravel - Visão Geral (Overview)

Gil Eduardo de Andrade

Conceitos Preliminares
(https://laravel.com/)

Introdução - Laravel
O Laravel é um framework voltado a plataforma web, que busca facilitar o desenvolvimento de aplicações, simplificando tarefas comuns como roteamento, autenticação, sessões e cache. O Laravel tem como objetivo tornar o processo de desenvolvimento agradável para o desenvolvedor, sem gerar impacto negativo sobre as funcionalidades da aplicação.
A aula proposta neste documento tem como objetivo apresentar a estrutura geral do framework, bem como seus principais módulos, cobrindo desde a criação de projetos, até os conceitos fundamentais de Rotas, Controllers, Views (com Blade) e Models (com Eloquent).

Instalação Composer - Gerenciador de Dependência
(https://getcomposer.org/download/)

Baixando e Configurando o Composer
(No terminal de comandos)

```shell
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"

php -r "if (hash_file('sha384', 'composer-setup.php') === 'c8b085408188070d5f52bcfe4ecfbee5f727afa458b2573b8eaaf77b3419b0bf2768dc67c86944da1544f06fa544fd47') { echo 'Installer verified'.PHP_EOL; } else { echo 'Installer corrupt'.PHP_EOL; unlink('composer-setup.php'); exit(1); }"

php composer-setup.php

php -r "unlink('composer-setup.php');"
```

Criando Projeto Laravel com Composer
(No terminal de comandos)

```Shell
php composer.phar create-project laravel/laravel laravel-project
```

Executando Projeto Laravel Criado
(No terminal de comandos)

```Shell
cd laravel-project

php artisan serve --port 8000
```

Acessando o Projeto Laravel
(No Navegador)

http://localhost:8000/

Repositório - Laravel 13 + MySQL + phpMyAdmin (Docker)

https://github.com/Instituto-Federal-PR/lavarel-13-mysql-phpmyadmin

Clonando Repositório
(No terminal de comandos)

```Shell
git clone https://github.com/Instituto-Federal-PR/laravel-13-mysql-phpmyadmin

cd laravel-13-mysql-phpmyadmin
```

Recriando o .env
(No terminal de comandos)

```Shell
cp .env.example .env
```

Subindo os Serviços e Recriando a Chave da Aplicaçãp

(No terminal de comandos)

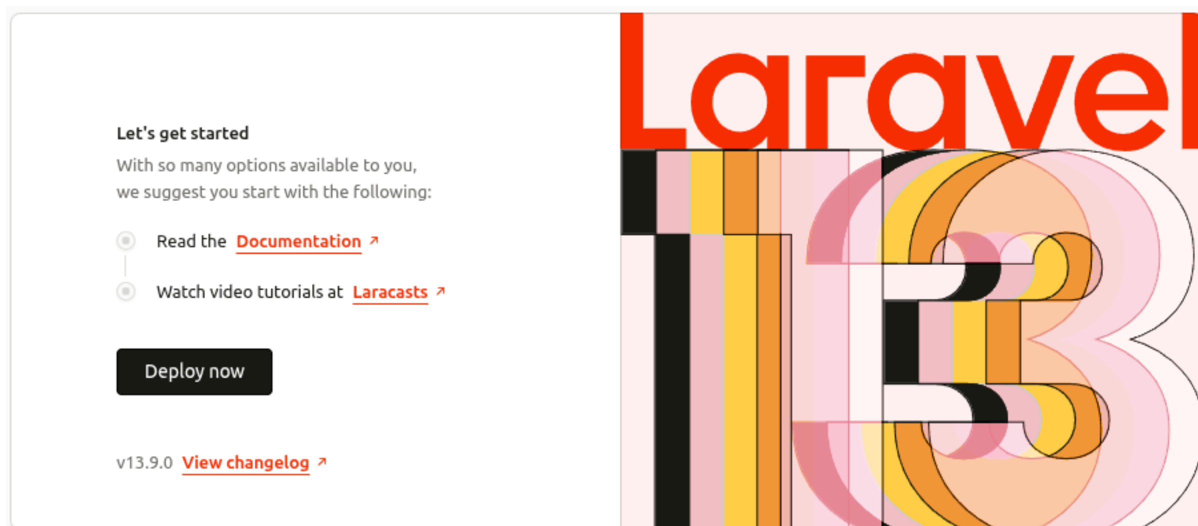
```shell
docker-compose up

docker-compose exec app php artisan key:generate
```

Efetuando a Primeira Migração

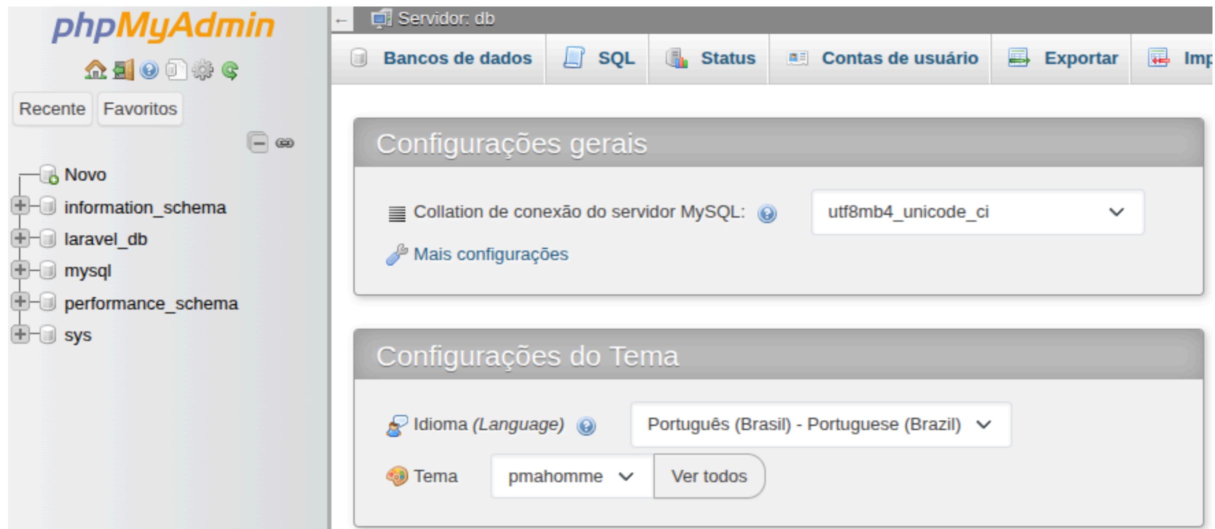
(Em um novo terminal, na mesma pasta)

```shell
docker-compose exec app php artisan migrate
```

Acessando a Aplicação: http://localhost:15000/



Acessando o phpMyAdmin: http://localhost:15001/



Rotas (https://laravel.com/docs/13.x/routing)

As rotas no Laravel são responsáveis por direcionar as requisições HTTP para os handlers apropriados. Todas as rotas são definidas nos arquivos de rota, localizados no diretório ***"routes"***.

Arquivos de Rotas

- **routes/web.php**: rotas que utilizam estado de sessão, proteção CSRF e cookies. Geralmente, são as rotas acessadas por um navegador.
- **routes/api.php**: rotas stateless, destinadas a APIs. Essas rotas recebem automaticamente o prefixo /api.

Definindo Rotas

A forma mais básica para uma rota é aceitar uma URI e uma ***"closure"*** (função anônima):

Arquivo de Rotas - "web.php" (Closure)

```php
use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
```

```
    return 'Hello World';
});
```

É comum, no entanto, que as rotas apontem para métodos de um Controller:

Arquivo de Rotas - “web.php” (Método da Controller)

```php
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

Parâmetros de Rota

As rotas podem capturar segmentos da URI, que são passados como argumentos para a closure ou método do controller:

- Parâmetros Obrigatórios:

```php
Route::get('/user/{id}', ...)
```

- Parâmetros Opcionais: Route::get('/user/{name?}', ...)

```php
Route::get('/user/{name?}', ...)
```

Controllers (https://laravel.com/docs/13.x/controllers)

As Controladoras são classes responsáveis por agrupar a lógica de tratamento de requisições as quais são relacionadas. Em vez de definir toda a lógica de requisição em ***“closures”***, nos arquivos de rota, é possível organizá-la em classes de controle.

Criando Controllers

Para criar uma nova classe de controle, use o comando Artisan *“make:controller”*:

```shell
php artisan make:controller UserController

docker-compose exec --user $(id -u):$(id -g) app php artisan
make:controller UserController
```

Uma classe de controle básica pode ter a seguinte aparência:

```php
<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller
{
    public function show(string $id): View
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

Resource Controllers

Para aplicações que seguem o padrão RESTful para operações CRUD (Create, Read, Update, Delete), o Laravel oferece os Resource Controllers. Com uma única linha de código, você pode registrar todas as rotas necessárias para um recurso:

```shell
php artisan make:controller PhotoController --resource

docker-compose exec --user $(id -u):$(id -g) app php artisan
make:controller PhotoController --resource
```

Uma classe de controle de recursos (Resource Controller) tem a seguinte aparência, onde os métodos de CRUD já são definidos na sua criação :

PHP
```php
namespace App\Http\Controllers;

class PhotoController extends Controller {

    /**
     * Display a listing of the resource.
     */
    public function index() { }

    /**
     * Show the form for creating a new resource.
     */
    public function create() { }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request) { }

    /**
     * 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) { }
```

```php
    /**
     * Remove the specified resource from storage.
     */
    public function destroy(string $id) { }
}
```

E no arquivo de rotas:

```php
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

Views e Blade Templates

(https://laravel.com/docs/13.x/views) (https://laravel.com/docs/13.x/blade)

As ***"views"*** são responsáveis por separar a lógica da sua aplicação da sua lógica de apresentação. Elas são armazenadas no diretório ***"resources/views"*** e utilizam a poderosa engine de templates Blade.

Criando e Retornando Views

Uma visualização pode ser criada como um arquivo ***".blade.php"*** (dentro da pasta ***"resources/views/"***) e retornada por uma rota ou *controller*:

```php
// Em uma rota ou controller
return view('greeting', ['name' => 'James']);
```

Blade Templates

O Blade é a engine de templates do Laravel. Diferente de outras, ela não restringe o uso de código PHP puro. Todos os templates Blade são compilados para código PHP puro e cacheados, o que significa que não há sobrecarga de performance.

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

Diretivas Blade

O Blade oferece diretivas para estruturas de controle comuns:

- ***Condicionais:*** @if, @elseif, @else, @endif
- ***Loops:*** @for, @foreach, @while
- ***Inclusão de Subviews:*** @include('shared.errors')

Layouts com Blade

O Blade facilita a criação de layouts consistentes através da herança de templates:

Layout Principal (layouts/app.blade.php)

```php
<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

View - Filha

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

```
@section('title', 'Page Title')

@section('content')
    <p>This is my body content.</p>
@endsection
```

Migrations (https://laravel.com/docs/13.x/migrations)

As migrações são como um controle de versão para a base de dados, permitindo que a equipe de desenvolvimento defina e compartilhe a definição do esquema do banco de dados da aplicação.

O que são Migrations?

As ***“migrations”*** do Laravel fornecem um método prático para criação e manipulação das tabelas em todos os sistemas de banco de dados suportados pelo Laravel. Cada arquivo de migração contém um timestamp no nome, permitindo ao Laravel determinar a ordem em que as migrações devem ser executadas.

Criando Migrations

Para gerar uma nova migration, use o comando Artisan ***“make:migration”***:

```shell
php artisan make:migration create_flights_table

docker-compose exec --user $(id -u):$(id -g) app php artisan make:migration create_flights_table
```

A nova migração será colocada no diretório ***“database/migrations”***. O Laravel tentará adivinhar o nome da tabela, e se a migração em questão criará uma nova tabela, baseado no nome da migração.

Estrutura de uma Migration

Uma classe de migração contém dois métodos: up e down. O método up é usado para adicionar novas tabelas, colunas ou índices ao banco de dados, enquanto o método down deve reverter as operações realizadas pelo método up.

PHP
```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

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

Tipos de Colunas Disponíveis

O Laravel oferece diversos tipos de colunas para construir suas tabelas:

- ***$table->id()*** - Alias de bigIncrements
- ***$table->string('name')*** - Coluna VARCHAR
- ***$table->text('description')*** - Coluna TEXT
- ***$table->integer('votes')*** - Coluna INTEGER
- ***$table->bigInteger('votes')*** - Coluna BIGINT
- ***$table->boolean('confirmed')*** - Coluna BOOLEAN
- ***$table->date('created_at')*** - Coluna DATE
- ***$table->dateTime('created_at')*** - Coluna DATETIME
- ***$table->decimal('amount', 8, 2)*** - Coluna DECIMAL com precisão e escala
- ***$table->timestamps()*** - Adiciona colunas created_at e updated_at
- ***$table->softDeletes()*** - Adiciona coluna deleted_at para soft deletes

Criando Chaves Estrangeiras

Para criar uma chave estrangeira, você pode usar o método foreign:

PHP
```php
Schema::create('posts', function (Blueprint $table) {
    $table->id();
    $table->unsignedBigInteger('user_id');
    $table->foreign('user_id')->references('id')->on('users');
    $table->string('title');
    $table->text('body');
    $table->timestamps();
});
```

Executando Migrations

Para executar todas as migrations pendentes:

Shell
```shell
php artisan migrate

docker-compose exec --user $(id -u):$(id -g) app php artisan migrate
```

Para verificar o status das migrations:

Shell
```shell
php artisan migrate:status
```

```
docker-compose exec --user $(id -u):$(id -g) app php artisan
migrate:status
```

Revertendo Migrations

Para reverter a última operação de migration:

```Shell
php artisan migrate:rollback

docker-compose exec --user $(id -u):$(id -g) app php artisan
migrate:rollback
```

Para reverter todas as migrations:

```Shell
php artisan migrate:reset

docker-compose exec --user $(id -u):$(id -g) app php artisan
migrate:reset
```

Para reverter todas as migrations e executá-las novamente:

```Shell
php artisan migrate:refresh

docker-compose exec --user $(id -u):$(id -g) app php artisan
migrate:refresh
```

Models e Eloquent ORM (https://laravel.com/docs/13.x/eloquent)

O Eloquent é o ORM (Object-Relational Mapper) incluído no Laravel. Ele fornece uma simples implementação do padrão ActiveRecord para trabalhar com seu banco

de dados. Cada tabela do banco de dados tem um "Model" correspondente que é usado para interagir com essa tabela.

Criando Models

Para criar um model, use o comando Artisan *"make:model"*:

```shell
php artisan make:model Flight

docker-compose exec --user $(id -u):$(id -g) app php artisan make:model Flight
```

Convenções do Eloquent

- **Nome da Tabela:** por padrão, o Eloquent assume que o nome da tabela é o plural do nome da ***"Model"*** em "snake case". Ex: ***Flight*** model > ***flights*** table.
- **Chave Primária:** o Eloquent assume que a chave primária é uma coluna com nome ***"id"*** auto-incrementável.
- **Timestamps:** o Eloquent espera que as colunas ***"created_at"*** e ***"updated_at"*** existam. Você pode desabilitar este comportamento com a propriedade $timestamps = false, dentro da sua "Model", ou removendo os dois campos dentro da migração.

Operações CRUD Básicas

Recuperando Dados:

```php
$flights = App\Models\Flight::all();
$flight = App\Models\Flight::find(1);
```

Inserindo Dados:

```php
$flight = new App\Models\Flight;
$flight->name = 'New Flight';
```

```
$flight->save();
```

Atualizando Dados:

```php
$flight = App\Models\Flight::find(1);
$flight->name = 'Updated Flight Name';
$flight->save();
```

Deletando Dados:

```php
$flight = App\Models\Flight::find(1);
$flight->delete();
```

Relacionamentos Eloquent (https://laravel.com/docs/12.x/eloquent-relationships)

As tabelas do banco de dados geralmente estão relacionadas umas às outras. Por exemplo, um post de blog pode ter muitos comentários, ou um pedido pode estar relacionado ao usuário que o fez. O Eloquent facilita o gerenciamento e trabalho com esses relacionamentos, suportando vários tipos comuns.

One to One (Um para Um)

Um relacionamento um-para-um é um tipo muito básico de relacionamento de banco de dados. Por exemplo, um modelo User pode estar associado a um modelo Phone.

Definindo o Relacionamento:

Classe de Modelo - User (hasOne)

```php
<?php

namespace App\Models;
```

```php
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Model
{
    /**
     * Obter o telefone associado ao usuário.
     */
    public function phone(): HasOne
    {
        return $this->hasOne(Phone::class);
    }
}
```

Classe de Modelo - Phone (belongsTo)

```php
class Phone extends Model
{
    /**
     * Obter o usuário que possui o telefone.
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
}
```

Usando o Relacionamento:

```php
$phone = User::find(1)->phone;
$user = Phone::find(1)->user;
```

ou

PHP
```php
$phone = User::with(['phone'])->find(1);
$user = Phone::with(['user'])->find(1);
```

One to Many (Um para Muitos)

Um relacionamento um-para-muitos é usado para definir relacionamentos onde um único modelo é pai de um ou mais modelos filhos. Por exemplo, um post de blog pode ter um número infinito de comentários.

Definindo o Relacionamento:

Classe de Modelo - Post (hasMany)

PHP
```php
class Post extends Model
{
    /**
     * Obter os comentários do post.
     */
    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```

Classe de Modelo - Comment (belongsTo)

PHP
```php
class Comment extends Model
{
    /**
     * Obter o post que possui o comentário.
     */
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```

Usando o Relacionamento:

```php
$comments = Post::find(1)->comments;

foreach ($comments as $comment) {
    echo $comment->body;
}

// Acessar o post de um comentário
$post = Comment::find(1)->post;
```

Many to Many (Muitos para Muitos)

Relacionamentos muitos-para-muitos são mais complexos que relacionamentos hasOne e hasMany. Um exemplo seria usuários que têm muitos papéis (roles), e esses papéis também são compartilhados por outros usuários.

Estrutura de Tabelas:

- users (id, name)
- roles (id, name)
- role_user (user_id, role_id) - Tabela pivot

Definindo o Relacionamento:

Classes de Modelo - User e Role (belongsToMany)

```php
class User extends Model
{
    /**
     * Os papéis que pertencem ao usuário.
     */
    public function roles(): BelongsToMany
    {
        return $this->belongsToMany(Role::class);
    }
}
```

```php
class Role extends Model
{
    /**
     * Os usuários que pertencem ao papel.
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class);
    }
}
```

Classes de Modelo - User e Role (belongsTo)

```php
class RoleUser extends Model
{
    public function role() {
        return $this->belongsTo(Role::class);
    }

    public function user() {
        return $this->belongsTo(User::class);
    }
}
```

Usando o Relacionamento:

```php
$user = User::find(1);

foreach ($user->roles as $role) {
    echo $role->name;
}

// Anexar papéis a um usuário
```

```
$user->roles()->attach($roleId);

// Remover papéis
$user->roles()->detach($roleId);

// Sincronizar papéis (remover todos e adicionar novos)
$user->roles()->sync([1, 2, 3]);
```