

Criar um CRUD com Laravel 12 e Bootstrap

Premissas

- Laravel 12 já instalado e configurado com Bootstrap (feito na aula passada).
- Banco de dados configurado e funcionando (XAMPP).

1 Criar projeto Laravel (caso ainda não tenha)

```
composer create-project laravel/laravel laravel-crud
```

Explicação: Cria um novo projeto Laravel completo na pasta laravel-crud. O Composer baixa todas as dependências do framework e configura a estrutura inicial.

Depois:

```
cd laravel-crud
```

Explicação: Entra na pasta do projeto para rodar comandos do Laravel, npm, etc.

2 Criar Banco de Dados usando o XAMPP

Iniciar o XAMPP e Servidor MySQL

- Abra o Painel de Controle do XAMPP no seu computador.
- Inicie o serviço MySQL clicando em "Start" ao lado dele.
- Certifique-se que o MySQL esteja rodando e que sua porta (geralmente 3306) esteja liberada.

Acessar o phpMyAdmin

- Abra seu navegador web e digite na barra de endereços:

```
http://localhost/phpmyadmin/
```

- O phpMyAdmin é uma interface web para gerenciar bancos MySQL facilmente.

Criar o Banco de Dados

- No phpMyAdmin, no menu lateral esquerdo, clique em "Novo" (ou "New").
- Na tela principal, em "Criar banco de dados", digite o nome do seu banco de dados.

laravel-crud

- Em **Collation**, escolha **utf8mb4_general_ci** para garantir suporte a caracteres em português e outros idiomas.
- Clique em "Criar".

Verificar o Banco Criado

- O banco aparecerá listado no painel à esquerda e você pode clicar no nome do banco para visualizá-lo. Inicialmente o banco está vazio, pronto para receber tabelas.

3 Configurar Banco de Dados no arquivo .env

Abra o arquivo **.env** que fica na raiz do projeto e configure as credenciais do seu banco MySQL:

```
DB_CONNECTION=mysql      # Tipo de banco de dados (MySQL)
DB_HOST=127.0.0.1        # Endereço do servidor do banco (localhost)
DB_PORT=3306              # Porta padrão do MySQL
DB_DATABASE=laravel-crud  # Substitua pelo nome do banco criado no MySQL
DB_USERNAME=root          # Usuário do MySQL (padrão root no XAMPP)
DB_PASSWORD=              # Senha do MySQL (normalmente vazia no XAMPP)
```

Explicação do código:

O Laravel usa essas variáveis para conectar-se ao banco. Configurar corretamente garante que as migrations, modelos e consultas funcionem.

É preciso alterar também as seguintes linhas, comentando a primeira linha e descomentando a segunda linha:

```
# APP_MAINTENANCE_DRIVER=file  
APP_MAINTENANCE_STORE=database
```

Para que serve o arquivo `.env` no Laravel:

- Armazena configurações sensíveis e específicas do ambiente, como credenciais de banco de dados, chaves de API, modo de debug, configuração de e-mail etc.
- Permite que você altere o comportamento da aplicação conforme o ambiente (local, produção, staging), mantendo o código-fonte limpo e sem hardcodes.
- O Laravel utiliza a biblioteca `vlucas/phpdotenv` para carregar essas variáveis automaticamente e disponibilizá-las via a função `env()`.
- Geralmente, o arquivo `.env` não deve ser versionado no controle de versões (como Git), para evitar exposição de dados sensíveis. No lugar, usa-se um arquivo `.env.example` para documentar quais variáveis precisam ser configuradas.

4 Criar Migration para a tabela courses

Gere a migration (estrutura/tabela no banco) com o comando Artisan:

```
php artisan make:migration create_courses_table --create=courses
```

Explicação: Esse comando cria o arquivo de migration dentro de `database/migrations/`, para definir a estrutura da tabela `courses`.

Abra o arquivo gerado:

(`database/migrations/xxxx_xx_xx_xxxxxx_create_courses_table.php`)

e modifique o método `up()` para:

```
public function up(): void  
{  
    Schema::create('courses', function (Blueprint $table) {  
        $table->id();  
        $table->string('name', 150)->unique();  
        $table->text('description')->nullable();  
        $table->timestamps();  
    });  
}
```

Explicação do código:

`public function up(): void`

O que faz:

Define o método `up()`, que é executado quando você roda o comando `php artisan migrate`. Sua função é aplicar (criar ou modificar) a estrutura do banco de dados.

Por que:

No ciclo das migrations, `up()` serve para “subir”/“aplicar” mudanças.

`Schema::create('courses', function (Blueprint $table) {`

O que faz:

Chama o método estático `create()` da facade `Schema` para criar uma nova tabela chamada `courses`. Recebe dois parâmetros: o nome da tabela (`'courses'`) e uma função anônima que recebe o objeto `$table` para definir as colunas.

Por que:

Esse é o padrão Laravel para construir tabelas novas de maneira programática e versionada.

`$table->id();`

O que faz:

Cria uma coluna chamada `id` do tipo `bigint unsigned`, que é auto-incrementável e será a chave primária da tabela.

Por que:

Praticamente todo registro em banco precisa de uma chave única para servir de identificador. No Laravel, o método `id()` já define tudo isso de modo seguro e eficiente.

`$table->string('name', 150)->unique();`

O que faz:

Cria uma coluna chamada `name` do tipo `VARCHAR` com limite de 150 caracteres. O método encadeado `->unique()` adiciona uma restrição de unicidade ao campo, ou seja, dois cursos não podem ter exatamente o mesmo nome.

Por que:

Controla/limita o tamanho do nome do curso e previne duplicidade, garantindo que o nome seja único (usado para buscas, exibição e referência).

`$table->text('description')->nullable();`

O que faz:

Define a coluna `description` como um campo `TEXT` (ideal para descrições maiores). O método `->nullable()` indica que o campo pode ficar em branco no banco, ou seja, não é obrigatório preencher a descrição ao criar um curso.

Por que:

Dá flexibilidade para ter cursos cadastrados mesmo sem descrição detalhada inicialmente.

`$table->timestamps();`

O que faz:

Cria automaticamente duas colunas: `created_at` e `updated_at`. O Laravel vai atualizar automaticamente essas datas quando o registro for criado ou alterado.

Por que:

Permite que você acompanhe quando cada registro foi adicionado e/ou editado, útil para auditoria e ordenação.

5 Executar as migrations

No terminal, execute o seguinte comando:

```
php artisan migrate
```

Explicação: O Laravel vai criar as tabelas no banco de acordo com todos os arquivos de migration pendentes. Esse é o comando obrigatório para que a tabela **courses** seja criada no banco de dados.

No Laravel, migrations são arquivos PHP que funcionam como um sistema de versionamento para o banco de dados da aplicação. Elas permitem que você crie, modifique e gerencie a estrutura das tabelas e colunas do banco de dados de forma programática, organizada e controlada.

Para que servem as migrations no Laravel?

- **Versionar o banco de dados:** Assim como o Git controla as versões do seu código, as migrations controlam as versões da estrutura do banco, garantindo que todos os desenvolvedores e ambientes tenham a mesma base de dados atualizada.
- **Criar e alterar tabelas e colunas:** Você define as mudanças estruturais (ex: criar tabela, adicionar coluna, mudar tipo) dentro da migration.
- **Facilitar a colaboração em equipe:** Com migrations, quando alguém faz mudanças na estrutura do banco, os demais apenas precisam rodar um comando para aplicar as alterações localmente.
- **Garantir segurança e organização:** Evita fazer alterações diretamente no banco, minimizando riscos e facilitando o controle das evoluções do esquema.
- **Reverter alterações:** Caso uma mudança cause problema, é possível ser desfeita com um comando.
- **Automatizar a criação do banco em novos ambientes:** Facilita criar o banco desde o zero em máquinas novas ou servidores.

Resumo do funcionamento

Cada migration é uma classe PHP contendo dois métodos:

up(): Define as alterações do banco que serão aplicadas (ex: criar uma tabela, adicionar colunas)

down(): Define como reverter essas alterações (ex: apagar a tabela, remover colunas)

Você cria as migrations com comando Artisan:

```
php artisan make:migration create_users_table --create=users
```

E aplica as alterações com:

```
php artisan migrate
```

Se precisar reverter a última alteração:

```
php artisan migrate:rollback
```

6 Criar Model Course

Para interagir com a tabela **courses** por meio do Eloquent ORM, crie a classe model:

```
php artisan make:model Course
```

Abra o arquivo `app/Models/Course.php` e configure o preenchimento em massa (**mass assignment**)

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Course extends Model
{
    use HasFactory;

    // Campos que podem ser preenchidos em massa (isso evita vulnerabilidades)
    protected $fillable = ['name', 'description'];
}
```

Explicação do código:

`<?php`

O que faz:

Inicia o arquivo PHP. Todo código PHP precisa começar com essa tag.

`namespace App\Models;`

O que faz:

Define o “namespace” da classe.

No Laravel, os Models ficam na pasta `app/Models` e usam o namespace correspondente.

Para que serve?

Organiza as classes, evita conflitos de nomes e facilita o autoload das classes pelo Composer.

`use Illuminate\Database\Eloquent\Model;`

`use Illuminate\Database\Eloquent\Factories\HasFactory;`

O que fazem:

`use Illuminate\Database\Eloquent\Model;`

Importa a classe base `Model` do Eloquent ORM do Laravel. Todos Models devem estender essa classe para receber os recursos de ORM (como consultas, relacionamento e mutators).

`use Illuminate\Database\Eloquent\Factories\HasFactory;`

Importa o trait `HasFactory`, que permite a geração de registros fake para esse Model usando Factories (útil para testes e seeders).

`class Course extends Model {`

O que faz:

Define a classe `Course` como um Model Eloquent.

Ela herda todos os métodos e propriedades da classe `Model`, automaticamente associando essa classe à tabela `courses` do banco (pelo padrão de convenção: nome da classe no singular, nome da tabela no plural).

```
use HasFactory;
```

O que faz:

Inclui o trait HasFactory na classe, habilitando o uso de Factories para criar dados fake do tipo Course.

Por que é importante?

Facilita testes e geração de dados automáticos em ambientes de desenvolvimento.

```
// Campos que podem ser preenchidos em massa (isso evita vulnerabilidades)
```

```
protected $fillable = ['name', 'description'];
```

O que faz:

Define o array \$fillable para informar ao Laravel quais atributos podem ser preenchidos em massa (mass assignment), ou seja, quando você faz Course::create(\$dados), só esses campos serão aceitos.

name: nome do curso (campo do banco)

description: descrição do curso

Por que é essencial?

Evita que campos não autorizados do banco sejam preenchidos por acidente ou por exploradores (segurança contra-ataques de mass assignment).

```
}
```

O que faz:

Fecha a definição da classe Course.

O que é o Eloquent ORM e para que serve:

O Eloquent representa cada tabela do banco de dados como uma classe (Model) no PHP, e cada registro dessa tabela como um objeto dessa classe.

Permite executar operações CRUD (Create, Read, Update, Delete) e consultas complexas de forma expressiva e legível, usando métodos PHP ao invés de SQL manual.

Utiliza convenções automáticas para associar nomes das tabelas, chaves primárias e timestamps, simplificando o trabalho do desenvolvedor.

Suporta relacionamentos entre tabelas (um-para-um, um-para-muitos, muitos-para-muitos, polimórficos etc.), permitindo trabalhar com dados relacionados facilmente.

Facilita o desenvolvimento orientado a objetos, mantendo o código limpo e aderente ao padrão MVC do Laravel.

Exemplo prático simplificado

Comparação entre usar o Laravel tradicional (DB facade) e o Eloquent ORM para buscar todos os usuários:

```
php
```

```
// Usando DB (query builder tradicional):
```

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

```
// Com Eloquent ORM:
```

```
$users = User::all();
```

Mais elegante, legível e fácil de estender: você manipula a **model User**, que representa a tabela users.

7 Criar Factory e Seeder para popular dados

Crie a classe CourseFactory:

```
php artisan make:factory CourseFactory --model=Course
```

Abra **database/factories/CourseFactory.php** e configure para gerar dados com **Faker**:

```
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use App\Models\Course;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Course>
 */
class CourseFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            // 'name' vai gerar uma frase curta com 3 palavras
            'name' => $this->faker->sentence(3),
            // 'description' cria um texto aleatório
            'description' => $this->faker->paragraph()
        ];
    }
}
```

Explicação do código:

<?php

O que faz:

Indica ao interpretador PHP que o arquivo contém código PHP. É obrigatório no início de cada arquivo de código.

namespace Database\Factories;

O que faz:

Define o namespace da classe. Database\Factories indica que esta factory está dentro do diretório database/factories, que é o local onde o Laravel mantém as factories padrão.

Por que é importante?

Organiza o código, evita conflitos de nomes de classes e permite o autoload correto via Composer.

```
use Illuminate\Database\Eloquent\Factories\Factory;
```

O que faz:

Importa a classe base Factory do Laravel. Essa classe fornece toda a estrutura para criar factories personalizadas — métodos, integração com Models e Faker, etc.

```
php
```

```
/**  
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Course>  
 */
```

O que é isso: Comentário de PHPDoc. Indica que esta factory é uma extensão de Factory configurada para trabalhar com o Model \App\Models\Course.

Para que serve?

Ajuda IDEs e ferramentas de análise estática a entender que essa factory gera instâncias de Course. Melhora o autocomplete e a verificação de tipos.

```
class CourseFactory extends Factory
```

O que faz:

Declara a classe CourseFactory, que herda de Illuminate\Database\Eloquent\Factories\Factory. Essa herança dá acesso ao método definition(), que define o estado padrão do Model.

```
/**  
 * Define the model's default state.  
 *  
 * @return array<string, mixed>  
 */
```

O que é:

Comentário de documentação (PHPDoc) explicando o propósito do método definition() e indicando que ele retorna um array associativo — usado internamente pelo Laravel para criar registros fake.

```
public function definition(): array
```

O que faz:

Define o método definition().

Papel no Laravel:

Sempre que a factory for usada, este método é chamado para gerar os dados padrões a serem inseridos no banco.

```
return [  
    // 'name' vai gerar uma frase curta com 3 palavras  
    'name' => $this->faker->sentence(3),  
    // 'description' cria um texto aleatório  
    'description' => $this->faker->paragraph()  
];
```

O que faz cada linha:

'name' => \$this->faker->sentence(3):

Usa o objeto \$this->faker (instância da biblioteca Faker) para gerar uma sentença aleatória contendo 3 palavras. Esse valor será atribuído à coluna name do curso. Exemplo de saída: "Introdução ao Laravel".

'description' => \$this->faker->paragraph():

Gera um parágrafo aleatório com texto fictício para simular a descrição do curso.

Exemplo de saída: "Este curso aborda conceitos fundamentais ...".

Por que usar Faker?

Facilita testes, pois você tem dados diferentes sempre que gerar. Rápido para popular tabelas no ambiente local. Simula uma base realista para desenvolver layouts, relatórios e funcionalidades.

Resumo do fluxo:

Quando você chama:

`Course::factory()->count(10)->create();`

O Laravel usa a `CourseFactory`, executa o método `definition()`, que retorna o array com `name` e `description` aleatórios e cria 10 registros na tabela `courses` com esses valores.

Crie a classe CourseSeeder:

```
php artisan make:seeder CourseSeeder
```

No arquivo `database/seeders/CourseSeeder.php`, adicione:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class CourseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        // Crie 10 cursos usando a factory
        \App\Models\Course::factory()->count(10)->create();
    }
}
```

Explicação do código:

```
<?php
```

O que é:

Indica que o arquivo contém código PHP. É obrigatório no começo de todo arquivo PHP.

```
namespace Database\Seeders;
```

O que é:

Define o namespace da classe, indicando que ela está na pasta database/seeders (o local padrão dos seeders no Laravel).

Por que importa:

Organiza o código de forma lógica. Permite o autoload pelo Composer sem conflitos de nomes entre classes.

```
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
```

```
use Illuminate\Database\Seeder;
```

O que são:

WithoutModelEvents: trait opcional que, se utilizada, desativa eventos de modelos (como creating, updating, etc.) durante a execução do seeder — útil para evitar efeitos colaterais quando populando o banco.

Seeder: Classe base que todo seeder herda. Contém o método call() para executar outros seeders e a definição do método run() que precisa ser implementado.

```
class CourseSeeder extends Seeder
```

O que é:

Declara a classe CourseSeeder como uma subclasse de Seeder.

Função:

Representa um seeder responsável apenas por popular dados para a tabela courses.

```
/**
* Run the database seeds.
*/
```

O que é:

Comentário PHPDoc. É apenas documentação explicando que este método executa a lógica para inserir os registros no banco.

```
public function run(): void
```

O que é:

Método obrigatório em todo seeder, chamado pelo Laravel quando você roda o comando:

```
php artisan db:seed --class=CourseSeeder
```

Função:

Implementar aqui as instruções para inserir dados no banco.

```
// Crie 10 cursos usando a factory
\App\Models\Course::factory()->count(10)->create();
```

O que faz:

\App\Models\Course::factory() — acessa a factory associada ao modelo Course (definida no arquivo database/factories/CourseFactory.php). Esta factory já sabe como gerar valores fictícios para os campos name e description.

->count(10) — define que serão criados 10 registros.

->create() — insere esses registros diretamente no banco de dados.

Por que é assim:

Usar factories dentro de seeders é uma forma rápida e padronizada de gerar dados de teste ou iniciais, sem precisar escrever manualmente valores fake.

}

O que é:

Fecha o método `run()`.

}

O que é:

Fecha a definição da classe `CourseSeeder`.

Agora registre a seeder criada em **database/seeders/DatabaseSeeder.php**:

```
$this->call(CourseSeeder::class);
```

Explicação do código:

`$this->call(CourseSeeder::class);`

Essa linha normalmente aparece dentro do método `run()` da classe `DatabaseSeeder` do Laravel.

Contexto: Classe DatabaseSeeder

No Laravel, a classe `DatabaseSeeder` (localizada em `database/seeders/DatabaseSeeder.php`) é o ponto de entrada padrão para a execução dos seeders — scripts que inserem dados iniciais no banco de dados.

Quando executamos no terminal:

```
php artisan db:seed
```

O Laravel chama o método `run()` do `DatabaseSeeder`. Dentro dele, usamos `$this->call()` para indicar quais seeders individuais devem ser executados.

Explicando a linha

```
$this->call(CourseSeeder::class);
```

1. \$this

Refere-se à instância atual da classe `DatabaseSeeder`, que herda da classe base `Illuminate\Database\Seeder`.

2. call()

É um método herdado da classe `Seeder`.

Função:

Executar um ou mais seeders passados como parâmetro.

Pode receber:

Uma classe de seeder ou um array de diversas classes seeding

Exemplo:

```
$this->call([
    CourseSeeder::class,
    UserSeeder::class
]);
```

3. CourseSeeder::class

`CourseSeeder` é uma classe seeder que você criou (ex.: `database/seeders/CourseSeeder.php`) para inserir dados na tabela `courses`. O `::class` retorna o nome totalmente qualificado da classe como uma string.

Exemplo:

"Database\Seeders\CourseSeeder"

É a forma mais segura e recomendada de passar o nome da classe para `$this->call()` (ao invés de usar string manualmente).

O que acontece quando essa linha é executada:

O Laravel localiza a classe CourseSeeder no namespace Database\Seeders.

- Instancia essa classe.
- Chama o método `run()` dela.
- Dentro do `run()` do CourseSeeder estará a lógica para popular a tabela — geralmente usando factories:

`\App\Models\Course::factory()->count(10)->create();`

Resultado: 10 registros fictícios de cursos são inseridos no banco.

Exemplo real de uso no DatabaseSeeder

```
public function run(): void
{
    // Aqui você pode chamar quantos seeders precisar
    $this->call(CourseSeeder::class); // Popula a tabela courses
    $this->call(UserSeeder::class); // Popula a tabela users
}
```

Quando você rodar:

`php artisan db:seed`

O Laravel vai executar o CourseSeeder e o UserSeeder na ordem definida.

Por fim, execute o comando para gerar os dados e após a execução verifique a tabela **courses**:

```
php artisan db:seed
```

Esse comando vai executar o seeder e inserir 10 registros na tabela courses. (explicação acima).

8 Criar Request para validação dos dados

Para centralizar regras de validação, crie uma classe Request:

```
php artisan make:request CourseRequest
```

No arquivo gerado `app/Http/Requests/CourseRequest.php`, defina as regras:

```
<?php
namespace App\Http\Requests;
```

```

use Illuminate\Foundation\Http\FormRequest;

class CourseRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string,
    \Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>
     */
    public function rules(): array
    {
        return [
            'name' => 'required|string|min:3|max:150', // obrigatório, string 3-150
caracteres
            'description' => 'nullable|string|min:10' // pode ser nulo, texto mínimo 10
caracteres
        ];
    }
}

```

Explicação do código:

namespace App\Http\Requests

Define que esta classe pertence ao namespace App\Http\Requests, padrão para classes de requisições personalizadas no Laravel.

use Illuminate\Foundation\Http\FormRequest

Importa a classe base FormRequest do Laravel, que estende a funcionalidade do request HTTP para incluir validações automáticas e autorização.

class CourseRequest extends FormRequest

Cria uma classe personalizada de requisição chamada CourseRequest, que herda de FormRequest. Essa herança traz métodos prontos para lidar com validação e autorização de dados enviados via HTTP.

Método authorize()

Define se o usuário está autorizado a fazer essa requisição. Retornar true significa que todos os usuários estão autorizados a realizar essa ação. Pode ser usado para implementar regras de permissão, restringindo quem pode enviar os dados.

Método rules()

Retorna um array associativo que define as regras de validação para cada campo enviado na requisição.
'name' => 'required|string|min:3|max:150'

O campo name é:

- obrigatório (required)
- deve ser do tipo string
- ter tamanho mínimo de 3 caracteres
- tamanho máximo de 150 caracteres

'description' => 'nullable|string|min:10'

O campo description é:

- opcional (nullable), pode ser omitido ou nulo
- se informado, deve ser do tipo string
- deve ter pelo menos 10 caracteres, se não for nulo

Essas regras garantem que os dados estejam no formato esperado antes de serem processados e salvos no banco.

Como a CourseRequest funciona na prática

Quando você o utiliza em métodos do controller (exemplo: store(CourseRequest \$request)), o Laravel automaticamente:

- Executa as regras de validação definidas em rules().
- Redireciona de volta ao formulário se houver erros, com mensagens detalhadas.
- Disponibiliza os dados validados via \$request->validated().

Caso a função authorize() retorne false, o Laravel bloqueia a requisição com erro 403 — útil para controle de acesso mais avançado.



Criar Controller Resource para CRUD

Use o Artisan para criar o controller com todos os métodos padrão:

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

Em **app/Http/Controllers/CourseController.php**, implemente os métodos usando o código abaixo:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Models\Course; // Model Course
use App\Http\Requests\CourseRequest; // Request customizado para validação

class CourseController extends Controller
```

```

{
    /**
     * Display a listing of the resource.
     * Lista paginada de cursos.
     */
    public function index()
    {
        $courses = Course::orderBy('created_at', 'desc')->paginate(5);
        return view('courses.index', compact('courses'));
    }

    /**
     * Show the form for creating a new resource.
     * Formulário para criar curso.
     */
    public function create()
    {
        return view('courses.create');
    }

    /**
     * Store a newly created resource in storage.
     * Armazenar dados no banco, com validação.
     */
    public function store(CourseRequest $request)
    {
        Course::create($request->validated());
        return redirect()->route('courses.index')->with('success', 'Curso criado com sucesso!');
    }

    /**
     * Display the specified resource.
     * Mostrar detalhes do curso.
     */
    public function show(Course $course)
    {
        return view('courses.show', compact('course'));
    }

    /**
     * Show the form for editing the specified resource.
     * Formulário de edição do curso.
     */
    public function edit(Course $course)
    {
        return view('courses.edit', compact('course'));
    }
}

```

```

    * Update the specified resource in storage.
    * Atualizar dados no banco com validação.
    */
public function update(CourseRequest $request, Course $course)
{
    $course->update($request->validated());
    return redirect()->route('courses.index')->with('success', 'Curso atualizado com sucesso!');
}

/**
 * Remove the specified resource from storage.
 * Remover curso.
 */
public function destroy(Course $course)
{
    $course->delete();
    return redirect()->route('courses.index')->with('success', 'Curso excluído com sucesso!');
}

```

Explicação do código

Namespace e Importações

namespace App\Http\Controllers;

Declara que esta classe está na pasta/namespaces de controllers padrão do Laravel.

use Illuminate\Http\Request;

Importa a classe Request padrão do Laravel. (Obs: aqui não está sendo usada, pois você trabalhou com CourseRequest para validação, mas geralmente está presente.)

use App\Models\Course;

Importa a Model Course, que representa a tabela courses no banco, permitindo acessar e manipular dados dos cursos via Eloquent (ORM do Laravel).

use App\Http\Requests\CourseRequest;

Importa a FormRequest customizada (classe que você gerou para centralizar regras de validação dos formulários de curso no lugar de validar dentro do controller).

Definição da Classe

class CourseController extends Controller

Define o controller que herda métodos utilitários da classe Controller padrão do Laravel. Controllers agrupam a lógica de requisições HTTP (ex.: acessar, criar, editar e deletar cursos).

Métodos

index()

Busca os cursos ordenando pelo mais recente primeiro (orderBy('created_at', 'desc')) e faz paginação (5 cursos por página).

`Course::(...)` usa a `model` para fazer consulta no banco. Retorna a view `courses.index` com a variável `$courses` disponível. No Blade, você mostra os cursos listados na tela e controla a paginação.

`create()`

Apenas retorna uma view contendo o formulário para criar um novo curso (sem dados, normalmente com campos vazios). Não faz consulta ao banco nesta etapa.

`store(CourseRequest $request)`

Recebe o formulário de criação de curso (POST).

O Laravel executa automaticamente a validação da requisição conforme definido na classe `CourseRequest`.

Se houver erro, retorna para o formulário; se passar, o array `$request->validated()` tem os dados seguros.

`Course::create(...)` insere um novo registro no banco (campos definidos como `fillable` na Model).

Redireciona para a lista de cursos (`courses.index`) e envia uma mensagem de sessão flash de sucesso, para exibir na tela.

`show(Course $course)`

Exibe os detalhes de um curso específico. O parâmetro `Course $course` usa Route Model Binding — o Laravel procura no banco pelo ID presente na URL e entrega a instância já carregada, ou lança erro 404 se não achar.

`edit(Course $course)`

Exibe o formulário de edição, já preenchido com os dados do curso indicado. Também usa Route Model Binding, tornando o código mais limpo e seguro.

`update(CourseRequest $request, Course $course)`

Recebe o envio do formulário de edição (PUT/PATCH). Os dados do request são validados pela `CourseRequest` e somente se válidos entram no update. Atualiza o registro do curso no banco e redireciona para a lista de cursos com mensagem de sucesso.

`destroy(Course $course)`

Remove do banco o curso correspondente ao objeto `$course` (que o Laravel já entrega via Route model binding) e redireciona para o index dos cursos mostrando mensagem de sucesso.

Uso de Model personalizada:

O Model centraliza toda manipulação de bases e consultas, permitindo trabalhar com objetos (ex: `$course->name`).

Uso de Request personalizada:

Separar toda a validação (ex: "nome é obrigatório, descrição min 10") para `CourseRequest`, deixando a controller limpa. Isso facilita manutenção e reutilização das regras.

Route Model Binding: O Laravel injeta a model correta baseado no id da rota/URL, economizando código e evitando erros comuns de consulta.

Paginação: Usar paginate facilita mostrar poucos cursos por vez e já fornece os links de navegação prontos para usar na view.

Mensagens de sessão: O `with('success', ...)` permite exibir avisos ao usuário após ações, melhorando a experiência.

Resumindo cada método:

- ***index(): lista paginada***
- ***create(): mostra formulário de novo***
- ***store(): cria novo validando dados***
- ***show(): mostra detalhes***
- ***edit(): mostra formulário para editar***
- ***update(): atualiza validando dados***
- ***destroy(): apaga curso***

Essa estrutura segue o padrão RESTful do Laravel, usando boas práticas (Request para validação e Model para dados), facilitando manutenção, segurança e produtividade.

Route Model Binding:

O Route Model Binding é um recurso do Laravel que faz a “injeção automática” de um objeto Model em métodos de controllers, com base no parâmetro passado na rota.

Como funciona?

Considere uma rota:

`Route::get('/courses/{course}', [CourseController::class, 'show']);`

{course} é um parâmetro dinâmico na URL que contém o ID (ou outro identificador) do curso.

Se o método do controller for:

```
public function show(Course $course)  
{ // ... }
```

O Laravel automaticamente busca o registro na tabela courses pelo ID recebido na URL. Se encontrar, instancia um objeto Course correspondente e o injeta no parâmetro \$course. Se não encontrar, retorna um erro 404 (não encontrado).

Benefícios:

- Código mais limpo: evita chamadas manuais para buscar o registro.
- Segurança: impede que IDs inválidos processados.
- Padronização: segue boas práticas do framework para manipular recursos RESTful.

1 0 Definir Rotas

No arquivo `routes/web.php`, adicione a rota padrão resource:

```
<?php  
  
use Illuminate\Support\Facades\Route;  
use App\Http\Controllers\CourseController;  
  
Route::get('/', function () {  
    return view('welcome');  
});  
  
Route::resource('courses', CourseController::class);
```

Explicação do código:

O que essa linha faz?

Essa linha registra, de uma só vez, todas as rotas REST (CRUD) necessárias para manipular recursos "courses" (cursos) usando o controlador CourseController. É uma das formas mais rápidas e organizadas do Laravel para criar rotas de CRUD.

Detalhamento dos Componentes

1. Route

Route é a facade do sistema de rotas do Laravel, responsável por registrar e gerenciar as URLs da sua aplicação e "linkar" cada URL a um método de um controller.

2. resource()

O método resource() é um atalho do Laravel que gera automaticamente um conjunto de 7 rotas padrões para operações CRUD (Create, Read, Update, Delete) em RESTful. Ele espera dois parâmetros:

- O primeiro, 'courses', define o prefixo da URL (será /courses).
- O segundo, CourseController::class, diz qual controller será responsável por essas rotas.

Quais rotas são criadas?

O Laravel cria automaticamente as seguintes rotas (com métodos HTTP e convenção de nomes):

Método HTTP	URL	Função	Método do Controller	Nome da rota
GET	/courses	Listar todos	index()	courses.index
GET	/courses/create	Formulário novo	create()	courses.create
POST	/courses	Salvar novo	store()	courses.store
GET	/courses/{course}	Ver um curso	show(\$course)	courses.show
GET	/courses/{course}/edit	Formulário edita	edit(\$course)	courses.edit
PUT/PATCH	/courses/{course}	Atualizar curso	update(\$course)	courses.update
DELETE	/courses/{course}	Deletar curso	destroy(\$course)	courses.destroy

Observações:

{course} é um parâmetro dinâmico (geralmente o ID do curso) que o Laravel injeta automaticamente via Route Model Binding. O método correspondente do controller será chamado conforme o HTTP e a URL.

Vantagens dessa abordagem:

- **Produtividade:** Não precisa definir cada rota manualmente.
- **Padrão:** Segue as boas práticas REST.
- **Automação:** O Laravel já associa cada método/página/formulário com o método correto do controller.
- **Manutenção simplificada:** Se mudar a ação do controller, não precisa mudar a rota.

Como acessar cada rota:

- *Listar todos: GET /courses*
- *Ver detalhes: GET /courses/1*
- *novo curso (form): GET /courses/create*
- *Salvar novo: POST /courses*
- *Editar (form): GET /courses/1/edit*
- *Atualizar: PUT/PATCH /courses/1*
- *Deletar: DELETE /courses/1*

No arquivo **app/Providers/AppServiceProvider.php**, altere-o e adicione as linhas abaixo:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Pagination\Paginator;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        //
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Paginator::useBootstrapFive();
    }
}
```

Usaremos o bootstrap para paginação na rota index para listar todos os registros, então, precisamos importá-lo no arquivo AppServiceProvider.php.

1 1 Criar as Views com Bootstrap via Artisan

Para criar as views, vamos usar os comandos abaixo:

```
php artisan make:view layouts.app  
php artisan make:view courses.index  
php artisan make:view courses.create  
php artisan make:view courses.edit  
php artisan make:view courses.show
```

Explicação:

php artisan make:view layouts.app

O que faz?

Cria o arquivo Blade chamado `app.blade.php` na pasta `resources/views/layouts/`.

Estrutura final:

`resources/views/layouts/app.blade.php`

Para que serve?

Esse arquivo costuma ser o layout principal da aplicação, por exemplo, base para outras views (`@extends('layouts.app')`).

O comando garante que o diretório existe, evita erros de nome e economiza tempo na estruturação inicial da view.

php artisan make:view courses.index

O que faz?

Gera a view `index.blade.php` na pasta `resources/views/courses/`.

Estrutura final:

`resources/views/courses/index.blade.php`

Para que serve?

Essa página é utilizada normalmente para listar todos os registros do CRUD de cursos. Com o comando, você tem rapidamente o arquivo correto para implementar a listagem.

php artisan make:view courses.create

O que faz?

Cria o arquivo `create.blade.php` dentro de `courses`.

Estrutura final:

`resources/views/courses/create.blade.php`

Para que serve?

Usado para exibir o formulário de criação de um novo curso no CRUD. O comando facilita criar a estrutura para essa funcionalidade, mantendo os nomes e localizações consistentes.

php artisan make:view courses.edit

O que faz?

Gera o arquivo `edit.blade.php` em `resources/views/courses/`.

Estrutura final:

`resources/views/courses/edit.blade.php`

Para que serve?

Usado para exibir o formulário de edição, pré-preenchido com os dados existentes de algum curso. Ganhamos tempo e evita erros ao automatizar a criação do arquivo.

php artisan make:view courses.show

O que faz?

Cria o arquivo `show.blade.php` em `resources/views/courses/`.

Estrutura final:

`resources/views/courses/show.blade.php`

Para que serve?

Utilizada para a visualização detalhada de um registro específico de curso (ex: acessado pela rota `/courses/{id}`).

Contexto dos comandos

Todos esses comandos utilizam o Artisan (CLI do Laravel) para criar arquivos de template Blade com o sufixo `.blade.php`.

Caso o diretório não exista (ex: `courses` ou `layouts`), o Laravel cria automaticamente o diretório apropriado. A nomenclatura do comando sempre segue o padrão “`pasta.nome`” — por exemplo, `courses.index` gera `resources/views/courses/index.blade.php`.

Esse recurso elimina o trabalho manual de abrir o editor, criar pastas e arquivos, agilizando o início de projetos ou manutenção de views.

Explicações e estrutura das views

Layout base: `resources/views/layouts/app.blade.php`

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
    <meta charset="UTF-8" />
    <title>@yield('title', 'Cursos')</title> {{-- Permite manter título dinâmico --}}
    @vite(['resources/sass/app.scss', 'resources/js/app.js']) {{-- Importa Bootstrap e
scripts --}}
</head>

<body>

    <div class="container mt-4">
        {{-- Exibe mensagem de sucesso, se houver --}}
        @if(session('success'))
            <div class="alert alert-success">{{ session('success') }}</div>
        @endif
    </div>
</body>
```

```

    {{-- área para conteúdo das views filhas --}}
    @yield('content')

```

```
</div>
```

```
</body>
```

```
</html>
```

Explicação do código:

Por que usar um layout base?

Usar um layout base no Laravel é fundamental para garantir organização, reutilização e consistência visual em todas as páginas da aplicação. Um layout base centraliza elementos comuns (como cabeçalho, navegação, scripts e rodapé), evitando repetição de código e facilitando a manutenção de grandes projetos.

Benefícios detalhados:

Reutilização de código: O layout base permite que todas as views herdem estruturas comuns (ex: <head>, menus, rodapé), eliminando duplicidade e facilitando atualizações globais. Qualquer mudança feita no layout reflete automaticamente em todas as páginas que o utilizam.

- **Consistência visual:** Garante que a aparência (cores, fontes, estrutura) permaneça idêntica em todas as páginas, promovendo experiência uniforme ao usuário.
- **Manutenção facilitada:** Com tudo centralizado, ajustes são feitos em um só lugar. Isto reduz tempo e risco de erro na manutenção, especialmente em grandes aplicações.
- **Organização modular:** Promove separação clara entre lógica de apresentação (layout/base) e lógica de cada página ou componente específico.
- **Facilidade para equipes:** Equipes podem trabalhar simultaneamente no layout e nas views específicas sem se atrapalhar, seguindo o padrão MVC e facilitando colaboração.
- **Performance e desempenho:** O Laravel Blade compila os layouts em PHP puro, armazena em cache e utiliza herança/slot, tornando a renderização mais rápida e eficiente.

Exemplos práticos:

Ao invés de repetir trechos como <nav>...</nav> e <footer>...</footer> em toda view, basta criar um layout base (`layouts/app.blade.php`) e nas views específicas usar:

```

@extends('layouts.app')
@section('content')
    
@endsection

```

Assim, o layout renderiza todo o cabeçalho, menus e rodapé automaticamente para cada página.

Em resumo, o layout base é a peça-chave para manter o código limpo, escalável, fácil de dar manutenção e com identidade visual uniforme em aplicativos Laravel modernos.

Página index - resources/views/courses/index.blade.php

Página para listagem dos cursos com paginação e ações CRUD.

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

@section('title', 'Lista de Cursos')

@section('content')

    <h1>Lista de Cursos</h1>

    <a href="{{ route('courses.create') }}" class="btn btn-primary mb-3">Adicionar
    Curso</a>

    <table class="table table-bordered">
        <thead>
            <tr>
                <th>Nome</th>
                <th>Descrição</th>
                <th>Ações</th>
            </tr>
        </thead>
        <tbody>
            @forelse($courses as $course)
                <tr>
                    <td>{{ $course->name }}</td>
                    <td>{{ \Illuminate\Support\Str::limit($course->description, 50) }}</td> {{-- Limita texto --}}
                    <td>
                        <a href="{{ route('courses.show', $course) }}" class="btn btn-info
                        btn-sm">Ver</a>
                        <a href="{{ route('courses.edit', $course) }}" class="btn btn-
                        warning btn-sm">Editar</a>

                        {{-- Form para deletar --}}
                        <form action="{{ route('courses.destroy', $course) }}"
method="POST" class="d-inline">
                            onsubmit="return confirm('Confirma exclusão?')"
                            @csrf
                            @method('DELETE')
                            <button class="btn btn-danger btn-sm"
type="submit">Deletar</button>
                        </form>
                    </td>
                </tr>
            @empty
                <tr>
                    <td colspan="3">Nenhum curso cadastrado.</td>
                </tr>
            @endfor
        </tbody>
    </table>
```

```

        </tr>
    @endforelse
</tbody>
</table>

{{-- Links de paginação --}}
{{ $courses->links() }}

@endsection

```

Explicação do código:

Essa view Blade é responsável por renderizar a lista de cursos no Laravel. Essa é uma view para o método index() do CourseController, e segue a convenção do Blade (template engine do Laravel).

@extends('layouts.app')

Função:

Define que este arquivo Blade herda um layout base chamado app.blade.php que está em resources/views/layouts/.

Por que usar:

Para aproveitar uma estrutura HTML e CSS padrão (cabeçalho, rodapé, etc.) presente no layout e evitar repetição de código.

@section('title', 'Lista de Cursos')

Função:

Define o conteúdo da seção title no layout pai (layouts.app), substituindo a marcação @yield('title') presente lá.

Resultado:

O título da aba/navegador será "Lista de Cursos".

@section('content')

Função:

Inicia a definição do conteúdo da seção content esperado pelo layout pai, onde será renderizado o conteúdo específico desta página.

<h1>Lista de Cursos</h1>

Função:

Título principal da página, exibido em destaque.

Adicionar Curso

Função:

Gera um link/botão (classe Bootstrap btn btn-primary) para a rota nomeada courses.create — que exibe o formulário de criação de curso.

Por que usar route():

Garante que o link seja construído com base nas rotas definidas no Laravel, evitando erros se a URL mudar.

```

<table class="table table-bordered">
  <thead>
    <tr>
      <th>Nome</th>
      <th>Descrição</th>
      <th>Ações</th>
    </tr>
  </thead>

```

Função:

Cria uma tabela estilizada pelo Bootstrap (*table table-bordered*) com cabeçalho (*thead*) contendo 3 colunas: Nome, Descrição, Ações.

```

<tbody>
  @forelse($courses as $course)

```

Função:

Itera sobre a variável *\$courses* (vinda do controller) usando a diretiva Blade *@forelse*. Se a coleção tiver itens, executa o loop, se estiver vazia, executa o bloco *@empty*.

```

    <tr>
      <td>{{ $course->name }}</td>
      <td>{{ \Illuminate\Support\Str::limit($course->description, 50)}}</td> {{-- Limita texto --}}

```

Função:

<td>{{ \$course->name }}</td>: mostra o nome do curso.

<td>{{ Str::limit(...) }}</td>: exibe a descrição, mas limitada a 50 caracteres para não quebrar o layout (o helper *Str::limit* é do Laravel).

```

    <td>
      <a href="{{ route('courses.show', $course) }}" class="btn btn-info btn-sm">Ver</a>
      <a href="{{ route('courses.edit', $course) }}" class="btn btn-warning btn-sm">Editar</a>

```

Função:

Link "Ver" → Rota para *courses.show* (detalhe do curso).

Link "Editar" → Rota para *courses.edit* (formulário para edição).

Ambos usam classes Bootstrap para cor e tamanho (*btn-sm* = pequeno).

```


<button type="button" class="btn btn-danger btn-sm"
  data-bs-toggle="modal"
  data-bs-target="#deleteModal{{ $course->id }}">
  Deletar
</button>

```

Comentário: Explica que o próximo botão servirá apenas para abrir o modal, não para excluir diretamente. *type="button"*: define que o botão não envia formulários (só executa ação de interface). Classes Bootstrap *btn btn-danger btn-sm*: estiliza como botão pequeno (*btn-sm*) e na cor vermelha (*btn-danger*).

data-bs-toggle="modal": atributo do Bootstrap que indica que este botão abre um modal.

data-bs-target="#deleteModal{{ \$course->id }}": indica o ID do modal que será aberto. Aqui usamos *{} \$course->id {}* para garantir que cada curso tenha um modal exclusivo, evitando conflitos se houver vários na página.

Texto "Deletar": informa a ação que o botão inicia.

```
<!-- Modal de confirmação -->
<div class="modal fade" id="deleteModal{{ $course->id }}"
    tabindex="-1"
    aria-labelledby="deleteModalLabel{{ $course->id }}"
    aria-hidden="true">
```

Comentário: Indica que o próximo bloco HTML é o modal.

<div class="modal fade">: classe modal cria um modal Bootstrap; fade adiciona efeito de transição suave ao abrir/fechar.

id="deleteModal{{ \$course->id }}": identifica unicamente esse modal (ligando-o ao botão via data-bs-target).

tabindex="-1": permite que o modal seja focável e fecha com a tecla ESC.

aria-labelledby="deleteModalLabel{{ \$course->id }}": acessibilidade — indica que o rótulo do modal é o elemento com esse id.

aria-hidden="true": define que o modal está invisível por padrão.

```
<div class="modal-dialog">
```

Define o container principal do modal, responsável pelo alinhamento e responsividade.

modal-dialog: controla largura e centralização; pode receber classes extras (modal-lg ou modal-sm) para variações de tamanho.

```
<div class="modal-content">
```

Container interno do modal que agrupa todo o conteúdo visual do mesmo: cabeçalho, corpo e rodapé. É onde o Bootstrap aplica o fundo branco, bordas e sombras padrão do modal.

```
<div class="modal-header">
    <h5 class="modal-title" id="deleteModalLabel{{ $course->id }}>Confirmar exclusão</h5>
    <button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Fechar"></button>
</div>
```

modal-header: barra do topo do modal.

<h5 class="modal-title">: título visível do modal; recebe id="deleteModalLabel{{ \$course->id }}" para vínculo com aria-labelledby.

Botão btn-close: ícone padrão para fechar; data-bs-dismiss="modal" fecha automaticamente o modal; aria-label melhora acessibilidade para leitores de tela.

```
<div class="modal-body">
```

*Tem certeza que deseja excluir o curso {{ \$course->name }}?
*

Esta ação não poderá ser desfeita.

```
</div>
```

modal-body: área central do modal, com a mensagem de confirmação.

Texto dinâmico exibe o nome do curso com **** para destacar.

Um **
** quebra a linha para a segunda frase de alerta ("Esta ação não poderá ser desfeita").

```
<div class="modal-footer">
```

<button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Cancelar</button>

<!-- Form para deletar -->

<form action="{{ route('courses.destroy', \$course) }}" method="POST" class="d-inline">

@csrf

@method('DELETE')

```

<button class="btn btn-danger" type="submit">Sim, excluir</button>
</form>
</div>
modal-footer: barra inferior com botões de ação.
Botão "Cancelar": data-bs-dismiss="modal" fecha o modal sem fazer nada; cor cinza (btn-secondary).
Form para deletar:
action="{{ route('courses.destroy', $course) }}": rota de exclusão do curso.
method="POST" com @method('DELETE'): Laravel converte para uma requisição DELETE.
@csrf {{-- Proteção contra CSRF --}}: Insere automaticamente um campo oculto com o token CSRF (Cross-Site Request Forgery) para proteger contra ataques cross-site request forgery, obrigatório no Laravel para validar que o formulário foi enviado pela própria aplicação.
Botão Sim, excluir: vermelho (btn-danger), envia o formulário confirmado a exclusão.

```

```

</div>
</div>
</div>
Fecham modal-content, o modal-dialog e o container externo do modal.

```

```

</tr>
@empty
<tr>
    <td colspan="3">Nenhum curso cadastrado.</td>
</tr>
@endforelse

```

Função:

Caso a lista \$courses esteja vazia, o trecho dentro de @empty será executado: uma linha única com mensagem "Nenhum curso cadastrado." e o colspan="3" faz a célula ocupar as 3 colunas da tabela.

```

</tbody>
</table>

```

Função:

Fecham o corpo e a tabela.

```

{{-- Links de paginação --}}
{{ $courses->links() }}

```

Função:

Renderiza links de paginação padrão do Laravel para navegar entre as páginas de resultados da coleção paginada \$courses.

@endsection

Função:

Encerra a seção content definida para ser injetada no layout base (layouts.app).

Essa view:

- Herda um layout base.
- Mostra o título e um botão para criar novo curso.
- Lista os cursos vindos do controller em forma de tabela.
- Adiciona botões de ação para ver, editar e deletar.

- *Mostra mensagem caso não haja cursos.*
- *Exibe paginação automática do Laravel.*

Página de criação de curso - `resources/views/courses/create.blade.php`

Página com o formulário para criar curso, com exibição de erros.

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

@section('title', 'Criar Curso')

@section('content')

    <h1>Criar Curso</h1>

    {{-- Formulário POST para adicionar novo curso --}}
    <form action="{{ route('courses.store') }}" method="POST">
        @csrf {{-- Proteção contra CSRF --}}
        @method('POST') {{-- Define método POST para inclusão --}}

        <div class="mb-3">
            <label for="name" class="form-label">Nome</label>
            {{-- Campo texto --}}
            <input type="text" name="name" value="{{ old('name') }}"
                   class="form-control @error('name') is-invalid @enderror" id="name" />
            {{-- Mensagem de erro --}}
            @error('name')
                <div class="invalid-feedback">{{ $message }}</div>
            @enderror
        </div>

        <div class="mb-3">
            <label for="description" class="form-label">Descrição</label>
            {{-- Área de texto --}}
            <textarea name="description" class="form-control @error('description') is-
invalid @enderror"
                      id="description">{{ old('description') }}</textarea>
            {{-- Mensagem de erro --}}
            @error('description')
                <div class="invalid-feedback">{{ $message }}</div>
            @enderror
        </div>

        <button type="submit" class="btn btn-success">Salvar</button>
        <a href="{{ route('courses.index') }}" class="btn btn-secondary">Voltar</a>
    </form>

@endsection
```

Explicação do código:

Essa view Blade é responsável por exibir o formulário de criação de um novo curso. Essa tela é normalmente usada pelo método create() do CourseController.

@extends('layouts.app')

Função:

Indica que esta view herda o layout base app.blade.php, localizado em resources/views/layouts/.

Impacto:

Permite que a página tenha a mesma estrutura visual de outras (cabeçalho, rodapé, CSS, scripts) sem duplicar código.

@section('title', 'Criar Curso')

Função:

Define o conteúdo da seção title do layout base, substituindo o @yield('title').

Impacto:

Define o título da aba do navegador como "Criar Curso".

@section('content')

Função:

Inicia a seção content, que será inserida no ponto apropriado do layout base, mostrado dentro de @yield('content') no layouts.app.

<h1>Criar Curso</h1>

Função:

Cabeçalho principal da página, indicando a ação que o usuário está realizando.

text

```
 {{-- Formulário POST para adicionar novo curso --}}
<form action="{{ route('courses.store') }}" method="POST">
```

Função:

Inicia um formulário HTML.

action="{{ route('courses.store') }}": Define que o formulário será enviado para a rota nomeada courses.store (padrão do resource do Laravel para salvar novo registro).

method="POST": Define que será um envio de dados para criar algo no servidor.

@csrf {{-- Proteção contra CSRF --}}

Função:

Insere automaticamente um campo oculto com o token CSRF (Cross-Site Request Forgery) para proteger contra ataques cross-site request forgery, obrigatório no Laravel para validar que o formulário foi enviado pela própria aplicação.

@method('POST')

Função:

Sobrecreve ou confirma o método HTTP. Aqui está redundante, pois já é POST, mas é útil para clareza e para manter padrão visual de código.

```

<div class="mb-3">
    <label for="name" class="form-label">Nome</label>
    {{-- Campo texto --}}
    <input type="text" name="name" value="{{ old('name') }}"
        class="form-control @error('name') is-invalid @enderror" id="name" />

```

Função:

Cria o campo de entrada de texto para o nome do curso.

value="{{ old('name') }}: Preenche o valor com dados antigos submetidos, caso haja erro de validação (não perde o que o usuário digitou).

class="form-control @error('name') is-invalid @enderror": Adiciona a classe CSS do Bootstrap `is-invalid` caso haja erro de validação para esse campo.

id="name": Associa o campo ao `<label for="name">`.

```

{{-- Mensagem de erro --}}
@error('name')
    <div class="invalid-feedback">{{ $message }}</div>
    @enderror
</div>

```

Função:

Se houver erro de validação no campo `name`, o Blade renderiza a mensagem `$message` envolta na classe Bootstrap `invalid-feedback`, para estilo padrão de erro.

```

<div class="mb-3">
    <label for="description" class="form-label">Descrição</label>
    {{-- Área de texto --}}
    <textarea name="description" class="form-control @error('description') is-invalid @enderror"
        id="description">{{ old('description') }}</textarea>

```

Função:

Campo de texto maior (`textarea`) para a descrição do curso. O `old('description')` mantém o texto digitado pelo usuário em caso de erro de validação. Se houver erro na validação, adiciona a classe `is-invalid` para destacar o campo.

```

{{-- Mensagem de erro --}}
@error('description')
    <div class="invalid-feedback">{{ $message }}</div>
    @enderror
</div>

```

Função:

Exibe mensagem de erro específica para a descrição, no formato padrão do Bootstrap.

```

<button type="submit" class="btn btn-success">Salvar</button>
<a href="{{ route('courses.index') }}" class="btn btn-secondary">Voltar</a>

```

Função:

Salvar: Botão que envia o formulário ao servidor.

Voltar: Link para a rota `courses.index` (lista de cursos), útil para desistir da operação sem enviar nada.

```
</form>
```

Função:

Fecha o formulário HTML.

@endsection

Função:

Encerra a seção `content` que foi aberta anteriormente, avisando ao Blade onde termina o conteúdo específico desta página.

 **Resumo da Função desta View**

Objetivo: Renderizar a página de criação de curso, com formulário protegido contra CSRF, validações visuais e mensagens de erro, mantendo valores já digitados em caso de falha.

Integração com Backend: Envia o formulário para a rota `courses.store`, que chama o método `store()` no `CourseController`. Usa a classe `CourseRequest` (no controller) para validar os campos `name` e `description`.

Frontend: Usa classes do Bootstrap para layout e estilo, aplica feedback visual para erros de validação e mantém consistência visual com o layout base.

Página de edição de curso - `resources/views/courses/edit.blade.php`

Página com o formulário semelhante ao de criação de curso, mas com dados preenchidos para editar.

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

@section('title', 'Editar Curso')

@section('content')

    <h1>Editar Curso</h1>

    <form action="{{ route('courses.update', $course) }}" method="POST">
        @csrf {{-- Proteção contra CSRF --}}
        @method('PUT') {{-- Define método PUT para atualização --}}

        <div class="mb-3">
            <label for="name" class="form-label">Nome</label>
            <input type="text" name="name" value="{{ old('name', $course->name) }}"
                   class="form-control @error('name') is-invalid @enderror" id="name" />
            @error('name')
                <div class="invalid-feedback">{{ $message }}</div>
            @enderror
        </div>

        <div class="mb-3">
            <label for="description" class="form-label">Descrição</label>
            <textarea name="description" class="form-control @error('description') is-
invalid @enderror"
                      id="description">{{ old('description', $course->description) }}</textarea>
            @error('description')
                <div class="invalid-feedback">{{ $message }}</div>
            @enderror
        </div>
    </form>
```

```

        </div>

        <button type="submit" class="btn btn-primary">Atualizar</button>
        <a href="{{ route('courses.index') }}" class="btn btn-secondary">Voltar</a>
    </form>

@endsection

```

Explicação do código:

Essa página segue o mesmo padrão de integração com o layout base e o uso de helpers/diretivas do Blade para formulários, segurança e tratamento de erros.

@extends('layouts.app')

Função:

Indica que esta view herda o layout app.blade.php, localizado em resources/views/layouts/.

Impacto:

Garante que a página tenha a mesma estrutura visual (HTML base, CSS, scripts, cabeçalho e rodapé) das outras páginas do sistema.

@section('title', 'Editar Curso')

Função:

Define o conteúdo da seção title definida no layout base, substituindo a tag @yield('title').

Impacto:

Altera o título exibido na aba do navegador para "Editar Curso".

@section('content')

Função:

Inicia o bloco de conteúdo que será inserido no ponto do layout onde existe @yield('content').

<h1>Editar Curso</h1>

Função:

Cabeçalho principal da página para indicar ao usuário que ele está na tela de edição.

<form action="{{ route('courses.update', \$course) }}" method="POST">

Função:

Inicia um formulário HTML.

O action="{{ route('courses.update', \$course) }}" indica que o formulário será enviado para a rota nomeada courses.update, passando o objeto \$course (o Laravel usa o id dele na URL).

method="POST" é o método HTML permitido, mas será transformado em PUT via @method (pois HTML puro não suporta PUT/PATCH diretamente).

@csrf {{-- Proteção contra CSRF --}}

Função:

Insere automaticamente um campo oculto com o token CSRF (Cross-Site Request Forgery) para proteger contra ataques cross-site request forgery, obrigatório no Laravel para validar que o formulário foi enviado pela própria aplicação.

```
@method('PUT') {{-- Define método PUT para atualização --}}
```

Função:

Informa ao Laravel que, embora o formulário seja enviado como POST, a intenção é processá-lo como uma requisição PUT (padrão REST para atualização de recursos).

```
<div class="mb-3">
  <label for="name" class="form-label">Nome</label>
  <input type="text" name="name" value="{{ old('name', $course->name) }}"
    class="form-control @error('name') is-invalid @enderror" id="name" />
```

Função:

Cria o campo de input para o nome do curso.

value="{{ old('name', \$course->name) }}": Define o valor do campo. Se houver erro de validação anterior, old('name') traz o último valor digitado, caso contrário, exibe o valor atual salvo no banco (\$course->name).
class="form-control @error('name') is-invalid @enderror": Aplica a classe is-invalid do Bootstrap se existir erro de validação para o campo name.

```
@error('name')
  <div class="invalid-feedback">{{ $message }}</div>
@enderror
</div>
```

Função:

Exibe a mensagem de erro para o campo name dentro de <div> com a classe invalid-feedback do Bootstrap, que formata e exibe a mensagem logo abaixo do input.

```
<div class="mb-3">
  <label for="description" class="form-label">Descrição</label>
  <textarea name="description" class="form-control @error('description') is-invalid @enderror"
    id="description">{{ old('description', $course->description) }}</textarea>
```

Função:

Campo de textarea para a descrição do curso.

{{ old('description', \$course->description) }} faz o mesmo princípio do campo name: mantém valor digitado anterior ou usa o existente no banco.

Aplica also a classe is-invalid se houver erro de validação.

```
@error('description')
  <div class="invalid-feedback">{{ $message }}</div>
@enderror
</div>
```

Função:

Exibe a mensagem de erro de validação para a descrição, usando a estilização padrão do Bootstrap.

```
<button type="submit" class="btn btn-primary">Atualizar</button>
<a href="{{ route('courses.index') }}" class="btn btn-secondary">Voltar</a>
```

Função:

Botão "Atualizar": envia o formulário, acionando o método update() no CourseController.

Botão "Voltar": link para a listagem de cursos (courses.index), permitindo ao usuário sair da tela de edição sem alterar nada.

`</form>`

Função:

Finaliza o formulário HTML.

`@endsection`

Função:

Finaliza a seção `content` aberta no começo, indicando ao Blade que o conteúdo específico desta view terminou e será injetado no layout base.

 **Resumo da Função desta View**

É a tela de edição de cursos. Recebe um objeto `$course` do backend e pré-preenche o formulário com seus dados atuais.

Usa:

- `CSRF` para segurança.
- `@method('PUT')` para seguir o padrão REST.
- `old()` para manter valores após erros de validação.
- `@error` para exibir erros de forma amigável.
- Classes do Bootstrap para estilização.

Página para mostrar os detalhes do curso - `resources/views/courses/show.blade.php`

Página com o formulário que exibe os detalhes completos do curso.

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

@section('title', 'Detalhes do Curso')

@section('content')

    <h1>Detalhes do Curso</h1>

    <ul class="list-group mb-3">
        <li class="list-group-item"><strong>Nome:</strong> {{ $course->name }}</li>
        <li class="list-group-item"><strong>Descrição:</strong> {{ $course->description }}</li>
    </ul>

    <a href="{{ route('courses.index') }}" class="btn btn-secondary">Voltar</a>

@endsection
```

Explicação do código:

Essa view é usada para exibir os detalhes de um único curso, tipicamente acionada pelo método `show()` do `CourseController` quando você acessa a rota `courses.show`.

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

Função:

Indica que esta view herda o layout base `app.blade.php`, localizado em `resources/views/layouts/`.

Impacto:

Reutiliza a estrutura HTML global da aplicação (cabeçalho, scripts, rodapé, estilos), evitando repetição de código e mantendo consistência visual.

```
@section('title', 'Detalhes do Curso')
```

Função:

Define o conteúdo da seção `title` do layout pai, substituindo o `@yield('title')` que existe nele.

Impacto:

Altera dinamicamente o título da aba do navegador para “`Detalhes do Curso`”.

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

Função:

Inicia a seção chamada `content`, que vai ser injetada no local correspondente do layout base, definido por `@yield('content')`.

```
<h1>Detalhes do Curso</h1>
```

Função:

Exibe um título principal (`H1`) na página, comunicando ao usuário o contexto da tela: ele está visualizando os detalhes de um curso específico.

```
<ul class="list-group mb-3">
```

Função:

Inicia uma lista não ordenada (``), estilizada com as classes Bootstrap:

list-group: aplica o estilo de lista do Bootstrap.

mb-3: adiciona uma margem inferior de 1 rem, para espaçamento visual.

```
<li class="list-group-item"><strong>Nome:</strong> {{ $course->name }}</li>
```

Função:

Cria um item da lista (``), estilizado com `list-group-item`.

Nome:: coloca o rótulo “Nome:” em negrito.

{{ \$course->name }}: exibe o valor do campo `name` do objeto `$course`, que foi passado pelo controller via Route Model Binding.

```
<li class="list-group-item"><strong>Descrição:</strong> {{ $course->description }}</li>
```

Função:

Outro item da lista, mostrando a descrição do curso.

Descrição:: coloca o rótulo “Descrição:” em negrito.

{{ \$course->description }}: conteúdo descritivo vindo do banco (tabela `courses`).

```
</ul>
```

Função:

Fecha a lista não ordenada.

```
<a href="{{ route('courses.index') }}" class="btn btn-secondary">Voltar</a>
```

Função:

Cria um link estilizado como botão, com a classe `btn btn-secondary` (estilo cinza do Bootstrap).

`href="{{ route('courses.index') }}": gera dinamicamente a URL para a rota nomeada courses.index (a listagem de cursos).`

Texto "Voltar": indica a ação de retornar para a tela anterior (lista geral de cursos).

`@endsection`

Função:

Finaliza a seção content aberta anteriormente. Tudo o que foi escrito entre `@section('content')` e `@endsection` será injetado no layout base no espaço reservado para conteúdo principal.

👉 Fluxo dessa view no Laravel

O usuário clica em “Ver” na listagem de cursos. O Laravel chama o método `show(Course $course)` no `CourseController`.

Esse método retorna:

```
return view('courses.show', compact('course'));
```

- Esta view recebe `$course` já carregado do banco (Route Model Binding).
- Renderiza as informações dentro de uma lista bonita usando Bootstrap.
- Exibe um link para voltar à listagem.

◆ Resumo dos conceitos aplicados:

- Herança de layout (`@extends`): reaproveita estrutura visual padrão.
- Seções (`@section`): envia conteúdo específico para posições definidas no layout pai.
- Interpolação Blade (`{{ }}`): exibe valores de variáveis de forma segura (escape de HTML para evitar XSS).
- Bootstrap: usado para estilizar lista e botão.
- Route Helper (`route()`): gera URLs baseadas nas rotas nomeadas, evitando dependência de URLs fixas.
- Route Model Binding: Laravel injeta automaticamente o curso correto na view.

1 2 Iniciar servidores backend e frontend para teste

Iniciar servidor Backend (Laravel)

No terminal, execute:

```
php artisan serve
```

Explicação:

O que faz este comando?

Inicia um servidor web embutido para desenvolvimento local do Laravel e permite que você rode sua aplicação Laravel sem a necessidade de configurar um Apache, Nginx ou outro servidor externo.

Por padrão, ao executar `php artisan serve`, será criado um servidor em:

`http://127.0.0.1:8000`

Como funciona?

Ele utiliza o servidor web embutido do PHP (iniciado a partir do PHP 5.4), responsável por tratar requisições HTTP durante o desenvolvimento.

Vantagens de usar php artisan serve:

- **Praticidade:** elimina a configuração manual de servidores web tradicionais para testes rápidos.
- **Portabilidade:** funciona em qualquer ambiente com PHP instalado.
- **Agilidade:** basta rodar o comando e programar, útil para equipes, novos projetos e prototipagem.

Personalizações possíveis:

O comando aceita argumentos para definir porta e host.

Alterar porta:

php artisan serve --port=8080

Aplicação rodará em <http://localhost:8080>.

Alterar host:

php artisan serve --host=192.168.0.10

Isso é útil se quiser acessar o sistema por outro IP na mesma rede.

Usar ambos:

php artisan serve --host=192.168.0.10 --port=9000

A aplicação ficará disponível em <http://192.168.0.10:9000>.

Iniciar o servidor Frontend (Vite)

Em outro terminal (ou na mesma aba após abrir outro terminal), execute:

```
npm run dev
```

Explicação:

O que é esse comando?

npm run dev é um comando de terminal usado em projetos Laravel modernos para compilar, empacotar e servir assets frontend (CSS, JS, imagens, etc.) no ambiente de desenvolvimento.

Ele utiliza ferramentas de build modernas, como **Vite (Laravel 9+)**, ou **Webpack/Laravel Mix (em versões anteriores)**, para processar e entregar arquivos otimizados do frontend de forma rápida e eficiente enquanto você desenvolve.

Como funciona na prática?

Empacotamento de assets:

- Compila arquivos SASS/SCSS, JavaScript, Typescript, Vue, React, etc.
- Junta (bundle) arquivos, converte recursos modernos para navegadores antigos e ajusta paths.
- Salva os arquivos finais (geralmente em `public/build` ou `public/js` / `public/css`).

Modo de "servidor de desenvolvimento":

- Inicia um servidor frontend (por padrão, rodando em `http://localhost:5173` com Vite) que "observa" alterações nos arquivos do frontend.
- Quando você edita um CSS, JS ou componente, ele recompila automaticamente e faz o hot reload, mostrando as mudanças instantaneamente no navegador sem precisar recarregar manualmente.

Integração com o Blade/Views:

No seu layout Blade, você usa diretivas como `@vite(['resources/js/app.js'])` e/ou `@vite(['resources/sass/app.scss'])`, e o Laravel conecta esses assets ao servidor frontend para garantir que você trabalhe sempre com a versão mais recente de seus arquivos.

Por que usar `npm run dev` no Laravel?

Desenvolvimento rápido: Mudanças no frontend aparecem quase instantaneamente.

Evita erros de cache: Sempre gera arquivos atualizados, sem risco de o navegador usar arquivos antigos.

Suporta recursos modernos: Pode usar JavaScript moderno, SASS, preprocessors, frameworks JS, etc. Tudo é convertido para funcionar em qualquer navegador.

Produtividade: Detecta e mostra erros de sintaxe e dependências imediatamente.

Testar no navegador

Para testar se está tudo certo, acesse:

`http://127.0.0.1:8000/courses` ou

`http://localhost:8000/courses`

Você deve ver a listagem de cursos. Clique em:

- **Adicionar Curso** para criar um novo.
- **Editar** para modificar um curso.
- **Ver** para ver detalhes do curso.
- **Deletar** para remover um curso (com confirmação).