Curso PHP

Do XLSX ao CMS (aula 9)

Os dados são preciosos
e duram mais que
os próprios sistemas
— Tim Berners-Lee

Sistema Gerenciador de Banco de Dados

SGBD

Em suma, um **SGBD** é o conjunto de programas de computador (softwares) responsáveis pelo gerenciamento de bases de dados.

O principal objetivo é retirar da aplicação cliente a responsabilidade de gerenciar o acesso, manipulação e organização dos dados.

Em bancos de dados relacionais a interface é constituída pelas APIs ou drivers do SGBD, que executam comandos na **linguagem SQL**.

(Datasus)

Structured Query Language

SQL

A **Linguagem de consulta estruturada (SQL)** é uma linguagem de programação para armazenar e processar informações em um banco de dados relacional.

Um banco de dados relacional armazena informações em formato tabular, com linhas e colunas representando diferentes atributos de dados e as várias relações entre os valores dos dados.

(AWS)

SQLite

SQLite

O SQLite é um banco de dados relacional que, diferentemente de outras ferramentas do tipo, não armazena informações em um servidor.

Essa independência acontece porque ele consegue colocar os seus arquivos dentro de si próprio.

(Ivan de Souza)

SQLite: instalação

Literalmente, você não faz uma instalação do SQLite. Você baixa o executável e realiza todas as operações a partir dali.

- https://sqlite.org/download.html

Caso precise de ajuda adicional, o guia abaixo mostrará como realizar o passo a passo nos mais diferentes sistemas operacionais:

- SQLite: da instalação até sua primeira tabela

PHP Data Object

PDO

No desenvolvimento de aplicações baseadas na linguagem PHP, surgiu a necessidade de juntar o acesso de diversas extensões de banco de dados presentes na linguagem, com isso surgiu o **PHP Data Object (PDO)**, que realiza também a abstração do banco de dados.

A sua vantagem está no objetivo de fornecer uma biblioteca limpa e consistente, para deixar unificadas as características das extensões que acessam os bancos de dados.

(Thiago)

Configurando o ambiente para PDO

```
composer require 'ext-pdo:*'
```

Inclui a extensão PDO nos requisitos do projeto

```
composer require 'ext-sqlite3:*'
```

Inclui a extensão SQLite nos requisitos do projeto

CRUD básico com PDO

Conexão

```php
<?php

try {
    // define conexão
    $pdo = new PDO("sqlite:database.sqlite");

    // define atributos
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);

} catch (PDOException $e) {

    // captura erros
    echo 'Error: ' . $e->getMessage();

}
```

Listar dados

```php
<?php

// requisita arquivo de conexão
require 'conexao.php';

// busca dados
$stmt = $pdo->query('SELECT ano, semestre FROM periodos');

// mostra dados
while($periodo = $stmt->fetch()) {
    echo "Período: {$ano}.{$semestre}\n";
}
```

Inserir registro

```php
<?php

// requisita arquivo de conexão
require 'conexao.php';

// prapara consulta
$stmt = $pdo->prepare("INSERT INTO periodos (ano, semestre) VALUES (:ano, :semestre)");

// definição de dados
$data = [
    'ano' => 2024,
    'semestre' => 1
];

// substitui valores
$stmt->bindParam(':ano', $data['ano'], PDO::PARAM_STR);
$stmt->bindParam(':semestre', $data['semestre'], PDO::PARAM_STR);

// executa query com substituição de valores
$stmt->execute();

// obtém chave primária do último registro inserido
$data['id'] = $pdo->lastInsertId();

// mostra registro
echo "O período #{$data['id']}: {$data['ano']}.{$data['semestre']}";
```

Inserir registro (simplificado)

```php
<?php

// requisita arquivo de conexão
require 'conexao.php';

// prapara consulta
$stmt = $pdo->prepare("INSERT INTO periodos (ano, semestre) VALUES (:ano, :semestre)");

// definição de dados
$data = [
    'ano' => 2024,
    'semestre' => 1
];

// substitui valores
// $stmt->bindParam(':ano', $data['ano'], PDO::PARAM_STR);
// $stmt->bindParam(':semestre', $data['semestre'], PDO::PARAM_STR);

// executa query com substituição de valores
$stmt->execute($data);

// obtém chave primária do último registro inserido
$data['id'] = $pdo->lastInsertId();

// mostra registro
echo "O período #{$data['id']}: {$data['ano']}.{$data['semestre']}";
```

Atualizar registro

```php
<?php

// requisita arquivo de conexão
require 'conexao.php';

// prapara consulta
$stmt = $pdo->prepare("UPDATE periodos SET semestre=:semestre WHERE id=:id");

// definição de dados
$data = [
    'id' => 1,
    'semestre' => 'A'
];

// executa query com substituição de valores
$stmt->execute($data);

// verifica se houve registros atualizados
if ($stmt->rowCount()) {
    echo "Registro atualizado";
} else {
    echo "Nada mudou.";
}
```

Excluir registro

```php
<?php

// requisita arquivo de conexão
require 'conexao.php';

// prapara consulta
$stmt = $pdo->prepare("DELETE FROM periodos WHERE id=:id");

// definição de dados
$data = [
    'id' => 1,
];

// executa query com substituição de valores
$stmt->execute($data);

// verifica se houve registros atualizados
if ($stmt->rowCount()) {
    echo "Registro apagado";
} else {
    echo "Nada mudou.";
}
```

Modelo de dados

Um modelo de dados é um modelo abstrato que organiza elementos de dados e padroniza como eles se relacionam uns com os outros e com as propriedades das entidades do mundo real.

(Wikipédia)

```php
<?php

class Periodo
{
    /** @var int */
    public $id;

    /** @var string */
    public $ano;

    /** @var string */
    public $semestre;
}
```

PDO::FETCH_CLASS

Ao definir o modo FETCH_CLASS, podemos referenciar uma classe como resultado da consulta, preenchendo automaticamente as propriedades presentes nela. Contudo, este método é de mão única, ou seja, apenas ao trazer os dados do banco de dados (SELECT), mas não disponível para levar para o banco de dados (INSERT, UPDATE).

No cenário de levar informações ao banco de dados, necessitamos criar métodos para interceptar os valores das propriedades para as colunas correspondentes.

```php
<?php

// define conexão PDO
$pdo = new PDO('sqlite:database.sqlite');

// prepara a consulta
$stmt = $pdo->prepare('SELECT * FROM periodos WHERE id=:id');

// define o modo de dados
$stmt->setFetchMode(PDO::FETCH_CLASS, Periodo::class);

// executa a query
$stmt->execute(['id' => 1]);

while ($periodo = $stmt->fetch()) {
    echo "Período #{$periodo->id}: {$periodo->ano}.{$periodo->semestre}";
}
```

O problema de usar PDO

É necessário escrever as queries de forma crua, dando margem a erros de digitação e outros fatores.

Para obter uma abstração, é necessário criar um arquivo intermediário para concentrar instruções mais básicas.

Caso precise de instruções mais elaborada, terá que realizar novas adaptações, o que nem sempre é algo fácil de manter.

Object-Relational Mapper
Object-Relational Mapping

ORM

Object-Relational Mapping (ORM), em português, mapeamento objeto-relacional, é uma técnica para aproximar o paradigma de desenvolvimento de aplicações orientadas a objetos ao paradigma do banco de dados relacional.

O uso da técnica de mapeamento objeto-relacional é realizado através de um mapeador objeto-relacional que geralmente é a biblioteca ou framework que ajuda no mapeamento e uso do banco de dados.

(Elton Fonseca)

Sem clubismo.
Porém, usaremos Eloquent.

Eloquent

Com base nos preceitos do ORM, **o Eloquent** abstrai toda a complexidade da interação com os bancos de dados utilizando *Models* para interagir com cada tabela.

(DialHost)

O Eloquent é parte do **framework Laravel**.

Um framework é um conjunto de ferramentas voltadas para atender diversas necessidades de um sistema.

`composer require illuminate/database`

Instala a dependência

Eloquent: configurações

As configurações de todas as dependências devem residir em um local específico e de fácil acesso. Para tal, usaremos o diretório **config**.

Cada parte do sistema terá um arquivo de configuração individual. Desta forma, ao modificar um arquivo, não interferimos em outro e, ao realizar o versionamento, apenas o arquivo modificado fica em evidência.

Para Eloquent, usaremos o arquivo **database.php**.

CRUD básico com Eloquent

Definição

```php
<?php

use Illuminate\Database\Eloquent\Model;

class Periodo extends Model
{
    /** @var int */
    public $id;

    /** @var string */
    public $ano;

    /** @var string */
    public $semestre;

    /** @var array */
    public $fillable = ['ano', 'semestre'];
}
```

Listar dados

```php
<?php

// buscar dados
$periodos = Periodo::select('ano', 'semestre')->get();

// mostrar dados
foreach ($periodos as $periodo) {
    echo "Período {$periodo->ano}.{$periodo->semestre}\n";
}
```

Inserir registro

```php
<?php

// prepara consulta
$periodo = new Periodo;

// definição de dados
$data = [
    'ano' => 2024,
    'semestre' => 1
];

// substitui valores
$periodo->fill($data);

// executa query
$periodo->save();

// mostra registro
echo "O período #{$periodo->id}: {$periodo->ano}.{$periodo->semestre}";
```

Inserir registro (simplificado)

```php
<?php

// definição de dados
$data = [
    'ano' => 2024,
    'semestre' => 1
];

// executa query com substituição de valores
$periodo = Periodo::create($data);

// mostra registro
echo "O período #{$periodo->id}: {$periodo->ano}.{$periodo->semestre}";
```

Atualizar registro

```php
<?php

// definição de dados
$data = [
    'id' => 1,
    'semestre' => 'A'
];

// localiza registro
// executa query com substituição de valores
$wasChanged = Periodo::where('id', $data['id'])->update($data);

// verifica se houve registros atualizados
if ($wasChanged) {
    echo "Registro atualizado";
} else {
    echo "Nada mudou.";
}
```

Excluir registro

```php
<?php

// definição de dados
$data = [
    'id' => 1,
];

// localiza registro
// executa query com substituição de valores
$wasDeleted = Periodo::where('id', $data['id'])->delete();

// verifica se houve registros atualizados
if ($wasDeleted) {
    echo "Registro apagado";
} else {
    echo "Nada mudou.";
}
```

Problemas de usar Eloquent

Apesar de não ser necessário escrever queries manualmente, consultas complexas podem ser um problema, caso não tenha clareza de como o ORM trabalha.

Queries com valores cru (raw queries) podem comprometer a integridade da consulta, abrindo margem para injeção de SQL.

Consultas complexas podem resultar em sobrecarga à memória, gerando gargalos e travamentos, quando não forem tratados corretamente (eager loading).

Migrations

Database migrations

Migrations trabalha na manipulação da base de dados:

- Criando
- Alterando
- Removendo

Uma forma de controlar as alterações do seu banco juntamente com o versionamento de sua aplicação.

(Carlos Eduardo)

SQL ou PHP?

Apesar de SQL ser uma boa ideia, os arquivos precisam estar na ordem correta para execução.

A query também precisa contar com um meio de reversão, caso seja necessário um “rollback”.

Como organizar tudo isso?

Uma boa prática para versionar o banco de dados, é escrever as migrations utilizando a própria linguagem utilizada no desenvolvimento.

```
migrations/
├── up/
│   ├── 001_create_users_table.sql
│   ├── 002_add_email_column_to_users.sql
│   └── 003_create_posts_table.sql
└── down/
    ├── 003_drop_posts_table.sql
    ├── 002_remove_email_column_from_users.sql
    └── 001_drop_users_table.sql
```

Migrations com Eloquent

Como dito antes, o Eloquent é parte do framework Laravel, portanto, possui comandos próprios para criar/gerenciar arquivos.

Neste projeto, usaremos somente as partes necessárias do framework para fins de entendimento do conceito.

Assim, criaremos comandos personalizados que nos permitam manipular os arquivos, tal como faríamos com Laravel.

`composer db:create-table …`

Para criar uma migration que cria uma tabela

CREATE TABLE

Após executar o comando anterior, será criado um arquivo correspondente.

No método **up**, adicionamos as configurações de colunas da tabela. Graças ao Eloquent, não precisamos conhecer a fundo SQL.

Por padrão, o Eloquent adiciona o método **timestamps**. Este método cria as colunas *create_at* e *updated_at*. Elas servem para orientar comportamentos internos do ORM. Não iremos alterar este comportamento.

```php
public function up(): void
{
    DB::schema()->create('periodos', function (Blueprint $table) {
        $table->id();
        $table->string('ano');
        $table->string('semestre');
        $table->timestamps();
        $table->unique(['ano', 'semestre']);
    });
}
```

`composer db:alter-table …`

Para criar uma migration para alterar uma tabela

ALTER TABLE

Algumas vezes, será necessário adicionar realizar uma modificação em uma tabela, algo que não estava planejado quando o projeto foi elaborado. É normal a medida que o projeto cresce.

Para isto, existem migrations de modificação, que invocam o comando **ALTER TABLE** e executam as mudanças necessárias.

```php
public function up(): void
{
    DB::schema()->table('agendamentos', function (Blueprint $table) {
        $table->renameColumn('data', 'data_marcada');
    });
}
```

`composer db:migrate`

Executa arquivos de migrations

Migrando

Agora que os arquivos existem, precisamos enviar as informações para o banco de dados.

Será executado o método **up** de todos os arquivos. Neste momento, o ORM consulta o banco de dados para saber quais migrations já foram executadas. Caso ela já tenha sido, será ignorada.

```
> db:migrate
SKIP: 00001_create_migrations_table
SKIP: 00002_create_periodos_table
SKIP: 00003_create_disciplinas_table
SKIP: 00004_create_atividades_table
SKIP: 00005_create_agendamentos_table
OK:   00006_alter_agendamentos_table
```

`composer db:rollback`

Reverte última migration executada

Revertendo

Caso perceba que alguma migration precisa ser corrigida, é possível reverter o processo.

Para isto, o que deve ser feito, ou melhor, desfeito, deve está no método **down**.

Será perguntado se deseja desfazer a mudança. Se negado, o processo é interrompido.

Por padrão, as migrations devem revertidas na ordem inversa de criação, ou seja, da última para a primeira.

```php
public function down(): void
{
    DB::schema()->table('agendamentos', function (Blueprint $table) {
        $table->renameColumn('data_marcada', 'marcada');
    });
}
```

```
> composer db:rollback
Desfazer 00005_create_agendamentos_table? [s/N]:s
OK: 00005_create_agendamentos_table
Desfazer 00004_create_atividades_table? [s/N]:n
Rollback finalizado
```

Removendo tudo de uma única vez

`composer db:rollback silent`

Reverte todas migrations executadas

Database Factories

Factories

Ao testar seu aplicativo ou semeando seu banco de dados, talvez seja necessário inserir alguns registros em seu banco de dados.

Em vez de especificar manualmente o valor de cada coluna, as factories permitem definir um conjunto de atributos padrão para cada um dos seus modelos Eloquent usando fábricas de modelos.

(Laravel)

Fake data

A principal função de uma factory é não precisar ficar pensando em dados para inserir em determinado modelo. Imagine ter que popular 10.000 registro manualmente para ter uma enorme massa de dados.

Para evitar tal situação, existem geradores de dados, nomeadas de *fake data*.

Como o nome sugere, são dados gerados de forma aleatória para os mais variados formatos e atendendo a requisitos específicos.

```
composer require fakerphp/faker --dev
```

Instala a depedência

Criando factories

`composer db:factory …`

Cria factory para determinado model

Factory: definição

Observe que na definição da factory, não usamos dados específicos, mas sim dados ficticios.

Estes dados, como dito antes, são obtidos a partir da classe Faker.

```php
public function definition(): array
{
    return [
        'ano' => faker()->numberBetween(2020, 2030),
        'semestre' => faker()->numberBetween(1, 4),
    ];
}
```

Factory: factory

É possível usar uma factory como dependência de outra factory, garantindo que haja consistência nos dados.

Este tipo de uso é fundamental para integridade de uso de chaves estrangeiras, que necessariamente precisa existir no banco de dados, caso não, irá gerar um erro.

```php
public function definition(): array
{
    return [
        'periodo_id' => Periodo::factory(),
        'nome' => faker()->unique()->firstName(),
        'cor' => faker()->safeColorName(),
    ];
}
```

Seeders

Semeando

Um seed é um arquivo que diz como o banco de dados deve ser carregado inicialmente.

Neste arquivo, você irá apontar a ordem de preenchimento das tabelas, para que não haja erro na atribuição de chaves estrangeiras, por exemplo.

Em nosso sistema, algumas tabelas depende de outras, é o caso de **disciplina**, que requer um **período**. Se for em ordem alfabetica, haverá erro.

`composer db:seed`

Executa semeador do banco de dados

Semenado

Nosso seed está configurado para opções carregar opções padronizadas. Além disso, é possível adicionar informações específicas.

É importante observar que nosso seed leva o banco de dados para um estado primário, isto é, os dados anteriores são perdidos.

Em alguns casos, os seeders são incrementais, ou seja, mantêm os dados, contudo, para evitar colisões entre chaves primárias, não usaremos dessa forma.

```
> composer db:seed
Escreva mais um valor para atividades [debate, prova, seminário]:
Escreva mais um valor para períodos [2023.2, 2024.1, 2024.2, 2025.1]:
Escreva mais um valor para disciplinas de 2023.2 []:
Escreva mais um valor para disciplinas de 2024.1 []:
Escreva mais um valor para disciplinas de 2024.2 []:
Escreva mais um valor para disciplinas de 2025.1 []:
Até quantos agendamentos por período (30)?
Atividades: 3
Períodos: 4
Disciplinas: 14
Agendamentos: 45
```

Relacionamentos

Bancos de dados relacionais

Um banco de dados relacional é um conjunto de informações que organiza dados em relações predefinidas, em que os dados são armazenados em uma ou mais tabelas (ou "relações") de colunas e linhas, facilitando a visualização e a compreensão de como diferentes estruturas de dados se relacionam.

Os relacionamentos são uma conexão lógica entre diferentes tabelas, que se estabelecem com base na interação entre elas.

(Google Cloud)

Bancos de dados relacionais: vantagens

A principal vantagem do modelo de banco de dados relacional é que ele oferece uma maneira intuitiva de representar dados e facilita o acesso a pontos de dados relacionados.

Como resultado, os bancos de dados relacionais são usados com mais frequência por organizações que precisam gerenciar grandes quantidades de dados estruturados, desde o rastreamento de inventário até o processamento de dados transacionais para a geração de registros de aplicativos.

(Google Cloud)

Relacionando models

Relacionando models

Os relacionamentos com Eloquent são intuitivos.

Em muitos casos, precisamos saber apenas dois comando:

- **HasMany:** possui muitos
- **BelongsTo:** pertence a

```php
<?php

namespace App\Models;

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

class Atividade extends Model
{
    use HasFactory;

    protected $fillable = [
        'nome',
        'cor',
    ];

    public function agendamentos(): HasMany
    {
        return $this->hasMany(Agendamento::class);
    }
}
```

Relacionando models

De mesma forma, models que recebem um relacionamento, pode possuir outros relacionamentos agregados.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Disciplina extends Model
{
    use HasFactory;

    protected $fillable = [
        'periodo_id',
        'nome',
        'cor',
    ];

    public function periodo(): BelongsTo
    {
        return $this->belongsTo(Periodo::class);
    }

    public function agendamentos(): HasMany
    {
        return $this->hasMany(Agendamento::class);
    }
}
```

Relacionando models

Um relacionamento pouco usado, porém necessário muitas vezes, é a **HasManyThrough** (“possui muitos através de”).

Este relacionamento, permite usar um modelo intermediário para obter o modelo principal.

No exemplo, podemos obter as atividades de um período por intermédio da disciplina associada ao agendamento.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;
use Illuminate\Database\Eloquent\Relations\HasManyThrough;

class Periodo extends Model
{
    use HasFactory;

    protected $fillable = [
        'ano',
        'semestre',
    ];

    public function atividades(): HasManyThrough
    {
        return $this->hasManyThrough(Atividade::class, Disciplina::class);
    }

    public function disciplinas(): HasMany
    {
        return $this->hasMany(Disciplina::class);
    }
}
```