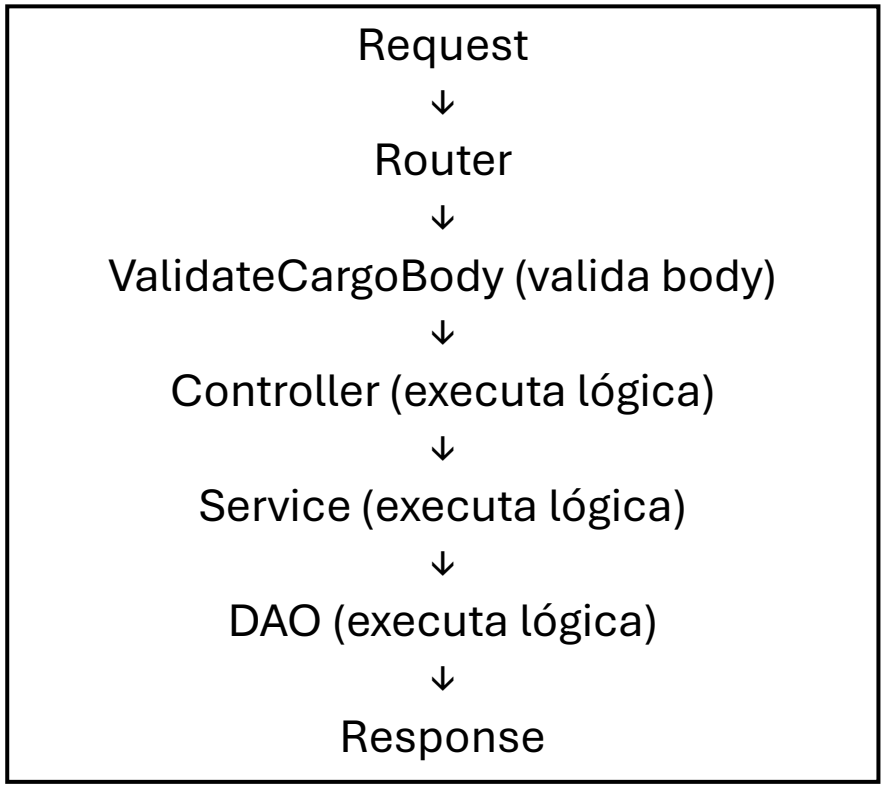
Exemplo prático: DI, Router, Middleware, Controller, Service, DAO e Model

Prof. Me. Hélio Lourenço Esperidião Ferreira

Sequência de processamento



Injeção de Dependência (Dependency Injection – DI)

A **injeção de dependência** é um padrão de projeto usado para reduzir o acoplamento entre classes.

Em vez de uma classe criar suas dependências, **essas dependências são fornecidas externamente**.

Ou seja: Em vez de a classe criar seus objetos, **alguém injeta esses objetos nela**.

- A injeção é feita via Construtor.

O que é DI (Injeção de Dependência)?

- Injeção de Dependência é um padrão onde você **não cria objetos dentro da sua classe**, mas sim **recebe eles de fora**.

```php
class MinhaClasse {
    private Database $db;
    public function __construct() {
        $this->db = new Database(); // forte acoplamento
    }
}
```

```php
class MinhaClasse {
    private Database $db;
    public function __construct($db) {
        $this->db = $db; // baixo acoplamento, objeto injetado via construtor
    }
}
```

DI no Slim: conceito geral

- O Slim Framework usa um container de dependências (geralmente baseado no PHP-DI).
- Esse container:
  - Cria objetos automaticamente
  - Resolve dependências
  - Injeta tudo para você
- Instale o Pacote
  - composer require php-di/php-di

Configurações importantes

```php
// 🔥 Habilita o Autowiring (injeção automática de dependências)
// Isso significa que o container vai tentar resolver automaticamente as
dependências das classes sem precisarmos configurar manualmente
$builder->useAutowiring(true);
```

```php
// 👉 Configuração MANUAL de dependências específicas
// Aqui definimos manualmente como criar o MysqlDatabase porque ele precisa
// de parâmetros de configuração (host, user, etc.) que não podem ser resolvidos
// automaticamente pelo Autowiring
// utilizamos quando queremos que uma instância seja utilizada sempre que necessário
e não varias instâncias.
$mysqlDatabase = new MysqlDatabase([
    'host' => 'localhost',
    'user' => 'root',
    'password' => '',
    'database' => 'gestao_rh',
]);

// Constrói o container com todas as configurações definidas
$container = $builder->build();

$container->set(MysqlDatabase::class, $mysqlDatabase);
```

Configurações importantes

```php
// 🔥 Registra a aplicação no container (PASSO ESSENCIAL)
// Isso torna a instância do Slim disponível para injeção em outras classes.
// Quando uma classe precisar do Slim\App, o container fornecerá esta instância
//a instância de App é utilizada nos roteadores, e deve ser a mesma instância
$container->set(\Slim\App::class, $app);
```

Utilize essa configuração toda vez que quiser que uma “única” instância seja utilizada toda vez que necessário e não múltiplas instências.

Router (Roteador)

O **Router** é responsável por **receber as requisições HTTP e direcioná-las para o Controller apropriado**.

Ele analisa a **rota da URL** e o **método HTTP** para decidir qual ação deve ser executada.

Funções do Router

- Receber requisições HTTP da aplicação
- Identificar a **URL da requisição**
- Verificar o **método HTTP** (GET, POST, PUT, DELETE)
- Encaminhar a requisição para o **Controller correspondente**
- Associar **middlewares** às rotas

O Router funciona como um **mapeador de rotas**, conectando **URLs da aplicação** às **ações dos Controllers**.

```
Request → Router → Middleware → Controller → Service → DAO(model) → Response
```

Roteador de cargo

```php
<?php
namespace Api\Routes;
use Slim\App;
use Api\Controllers\CargoController;
use Api\Middlewares\Cargo\ValidateCargoBody;
use Api\Middlewares\Cargo\ValidateCargoId;

class CargoRouter {
    private App $app;
    public function __construct(App $app){
        $this->app = $app;
    }
```

```php
public function setupRoutes(): void {
        $this->app->post(
            '/cargos’,
             [CargoController::class, 'createController']
        )->add(ValidateCargoBody::class);

        $this->app->get(
            '/cargos',
            [CargoController::class, 'findAllController']
        );

        $this->app->get(
            '/cargos/count',
            [CargoController::class, 'countController']
        );

        $this->app->get(
            '/cargos/{idCargo}',
            [CargoController::class, 'findByIdController']
        )->add(ValidateCargoId::class);

        $this->app->put(
            '/cargos/{idCargo}',
            [CargoController::class, 'updateController’]        //3º a executa
        )   ->add(ValidateCargoBody::class)                     //2º a executa
            ->add(ValidateCargoId::class);                      //1º a executa

        $this->app->delete(
            '/cargos/{idCargo}',
            [CargoController::class, 'deleteController']
        )->add(ValidateCargoId::class);
    }
}
```

```php
<?php class CargoRouter {
    private const BASE = '/api/v1/cargos';
    private App $app;
    public function __construct(App $app){$this->app = $app;}
    public function setupRoutes(): void {
        $this->app->post(self::BASE,
            [CargoController::class, 'createController'] // 3º )
            ->add(ValidateCargoBody::class)   // 2º
            ->add(JwtMiddleware::class);      // 1º

        $this->app->get(self::BASE,[CargoController::class, 'findAllController']) // 2º
            ->add(JwtMiddleware::class);// 1º (primeiro a executar)

        $this->app->get(self::BASE . '/Count', [CargoController::class, 'countController']) // 2
            ->add(JwtMiddleware::class);      // 1º (primeiro a executar) → JwtMiddleware (autenticação)

        $this->app->get(self::BASE . '/{idCargo}',[CargoController::class, 'findByIdController'] ) // 3º (último a executar)
            ->add(ValidateCargoId::class)   // 2º (segundo a executar
            ->add(JwtMiddleware::class);     // 1º (primeiro a executar)

        $this->app->put( self::BASE . '/{idCargo} ', [CargoController::class, 'updateController']   )
            ->add(ValidateCargoBody::class)
            ->add(ValidateCargoId::class)
            ->add(JwtMiddleware::class);

        $this->app->delete(
            self::BASE . '/{idCargo}', [CargoController::class, 'deleteController'])
            ->add(ValidateCargoId::class)
            ->add(JwtMiddleware::class);
}
```

Middleware

O **Middleware** é um componente intermediário que executa **lógicas antes ou depois da requisição chegar ao Controller**.

- Ele funciona como um **filtro ou interceptador** da requisição.

Funções comuns de um Middleware

- verificação de permissões (autorização)
- logs de requisições
- validação da estrutura de dados
- tratamento de erros
- controle de CORS (“quem” a aplicação responde)

```
Request → Router → Middleware → Controller → Service → DAO(model) → Response
```

Forma Profissional

Um middleware é representado por meio de uma classe.

Cada middleware deve ser responsável por uma validação especifica.

A classe deve possuir um nome intuitivo que representa a validação:

- ValidateCargoBody
- ValidateCargoId

A classe deve implementar a interface: MiddlewareInterface

- A classe deve implementar o método process devido a interface MiddlewareInterface

ValidateCargoBody

Esse middleware intercepta a requisição antes do controller, valida os dados obrigatórios e, se estiver tudo certo, deixa a execução continuar. Caso contrário, ele interrompe o fluxo lançando um erro.

```
Request → Validação → (Erro OU continua) → Próximo handler
```

ValidateCargoBody (Middleware)

- Classe responsável por **validar os dados da requisição HTTP**
- Atua **antes do controller**
- Implementa a interface `MiddlewareInterface` (PSR-15)
- Faz parte da **cadeia de middlewares**
- O método process valida a requisição

Função principal:

- Garantir que o corpo da requisição contém os dados obrigatórios do cargo

1. Obtém o corpo da requisição (`getParsedBody()`)
2. Verifica se existe o campo `cargo`
3. Verifica se `nomeCargo` está preenchido
4. Se erro → lança `ErrorResponse`
5. Se válido → chama `$handler->handle()`

Middleware: ValidateCargoBody

Estrutura Base de um middleware

```php
<?php
namespace Api\Middleware\Cargo;
use Psr\Http\Message\ServerRequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;
use Psr\Http\Server\RequestHandlerInterface as RequestHandler;
use Psr\Http\Server\MiddlewareInterface;
use Api\Utils\ErrorResponse;
class ValidateCargoBody implements MiddlewareInterface{

}
```

Método process

```php
public function process(Request $request, RequestHandler $handler): Response {
        // Lê o JSON bruto enviado no body
        $body = $request->getBody()->getContents();
        $objPHP = json_decode($body); // Converte JSON para objeto stdClass
        // Validação 1: verificar se o campo principal 'cargo' existe
        if (!isset($objPHP->cargo)) {
            throw new ErrorResponse(
                httpCode: 400,
                message: "Erro na validação de dados",
                error: [
                    "message" => "O campo 'cargo' é obrigatório!"
                ]
            );
        }
        $cargo = $objPHP->cargo;          // Armazena objeto cargo
       // Validação 2: verificar se 'nomeCargo' existe e não está vazio
        if (!isset($cargo->nomeCargo) || trim((string) $cargo->nomeCargo) === "") {
            throw new ErrorResponse(
                httpCode: 400,
                message: "Erro na validação de dados",
                error: [
                    "message" => "O campo 'nomeCargo' é obrigatório!"
                ]
            );
        }
        // Se tudo estiver válido, segue fluxo da requisição
        return $handler->handle($request);
    }
```

ValidateCargoId

```php
public function process(Request $request, RequestHandler $handler): Response {
        // Obtém o contexto da rota atual usando o objeto de requisição
        $routeContext = RouteContext::fromRequest($request);
        // Recupera a rota que está sendo chamada
        $route = $routeContext->getRoute();
        // Recupera os argumentos/parametros passados na rota (ex: /cargos/{idCargo})
        $routeArgs = $route->getArguments();
        // ------------------------------------------------------------
        // Validação 2: verificar se o parâmetro obrigatório 'idCargo' existe e não está vazio
        // ------------------------------------------------------------
        if (!isset($routeArgs['idCargo']) || $routeArgs['idCargo'] === "") {
            throw new ErrorResponse(
                httpCode: 400,
                message: "Erro na validação de dados",
                error: [
                    "message" => "O parâmetro 'idCargo' é obrigatório!"  // Mensagem detalhada
                ]
            );
        }
```

Como validar dados pelos parâmetros da rota? Em que verbos http são passados parâmetros pela rota?

Middleare de controle de erros: Classe server

- Nem sempre é necessário implementar o método process
- Middlwares globais podem ser feitos de outras formas
- Afim de capturar todos os erros possíveis, foi implementado um middleware na calsse Server.php

```php
    private function setupErrorHandling(): void{
        $errorMiddleware = $this->app->addErrorMiddleware(true, true, true);
        $errorMiddleware->setDefaultErrorHandler(
            function (ServerRequestInterface $request, \Throwable $exception)  {
                $response = new \Slim\Psr7\Response();
                $status = 500;
                if ($exception instanceof ErrorResponse) {
                    $payload = [
                        'success' => false,                             // Indica falha na operação
                        'message' => $exception->getMessage(),          // Mensagem amigável para o usuário
                        'error' => $exception->getError() ?? (object) [], // Detalhes adicionais do erro
                    ];
                    $status = $exception->getHttpCode();                // Status HTTP específico (ex: 404, 400, 403)
                }
                // CASO 2: ERRO INESPERADO (qualquer outra exceção)
               else {
                    $payload = [
                        'success' => false,                             // Indica falha na operação
                        'message' => $exception->getMessage(),          // Mensagem de erro técnica
                        'error' => [
                            'code' => $exception->getCode(),            // Código do erro
                            'sack' => $exception->getTrace(),
                            'file' => $exception->getFile(),            // Arquivo onde ocorreu
                            'line' => $exception->getLine(),            // Linha do erro
                        ],
                    ];
                    // Nota: Em produção, evite mostrar detalhes técnicos como file/line
                }
               $response->getBody()->write(json_encode($payload, JSON_PRETTY_PRINT | JSON_UNESCAPED_UNICODE));
                   return $response
                    ->withHeader('Content-Type', 'application/json')  // Indica que resposta é JSON
                    ->withStatus($status);                             // Define o status HTTP
            }
        );
    }
```

Controller

O **Controller** é responsável por:

- receber a requisição **encaminhada pelo Router (ou middleware se houver)**
- chamar os Services necessários
- retornar a resposta ao usuário

Características:

- pertence à camada de apresentação (faz comunicação com o frontend)
- não deve conter regras de negócio
- foca na **orquestração da entrada e saída de dados**

CargoController

O controller recebe a requisição, delega a lógica para o service e devolve uma resposta HTTP padronizada. Ele não contém regra de negócio, apenas coordena o fluxo.

CargoController (Controller da API)

- Responsável por controlar os recursos da entidade **Cargo**
- Atua como intermediário entre:
  - Router (rotas)
  - Service (regra de negócio)
- Implementa o padrão **MVC (Controller)**

Função principal:

- Receber requisição HTTP
- Chamar o serviço
- Retornar resposta padronizada (JSON)

Passos:

1. Recebe dados da requisição (`Request`)
2. Extrai informações (body, params)
3. Chama o `CargoService`
4. Monta resposta JSON
5. Retorna `Response` com status HTTP

`Request → Router → Middleware → Controller → Service → DAO(model) → Response`

Controller: CargoController

```php
<?php
namespace Api\Controllers;

// Importações das classes necessárias
use Psr\Http\Message\ResponseInterface as Response;  // Interface para respostas HTTP PSR-7
use Psr\Http\Message\ServerRequestInterface as Request;  // Interface para requisições HTTP PSR-7
use Api\Services\CargoService;  // Serviço de negócio para operações com cargos
use Api\Http\ErrorResponse;  // Classe para respostas de erro personalizadas

class CargoController {
    private CargoService $cargoService;
    public function __construct(CargoService $cargoServiceDependency)
    {
        // Armazena a dependência recebida na propriedade privada
        $this->cargoService = $cargoServiceDependency;
    }
```

Observe que a classe recebe como construtor uma instancia de `CargoService` isso porque para funcionar o controle depende de CargoService
por isso é recebido via injeção de dependência

Controller: createController

```php
public function createController(Request $request, Response $response, array $args): Response {
    $body = $request->getBody()->getContents();
    $objPHP = json_decode($body);
    $novoCargo = $this->cargoService->createService($objPHP);
    $resposta = [
        'success' => true,
        'message' => 'Cadastro realizado com sucesso',
        'data' => [
            'cargos' => [
                [
                    'idCargo' => $novoCargo->getIdCargo(),
                    'nomeCargo' => $novoCargo->getNomeCargo()
                ]
            ]
        ]
    ];
    $response->getBody()->write(json_encode($resposta));
    return $response
        ->withHeader('Content-Type', 'application/json')
        ->withStatus(201);
}
```

Controller: findAllController

```php
public function findAllController(Request $request, Response $response, array $args): Response {
    $cargos = $this->cargoService->findAllService();
    $resposta = [
        'success' => true,
        'message' => 'Busca realizada com sucesso',
        'data' => [
            'cargos' => $cargos
        ]
    ];
    $response->getBody()->write(json_encode($resposta));
    return $response
        ->withHeader('Content-Type', 'application/json')
        ->withStatus(200);
}
```

Controller: findByIdController

```php
public function findByIdController(Request $request, Response $response, array $args): Response {
      $idCargo = (int) $args['idCargo'];
      $cargo = $this->cargoService->findByIdService($idCargo);
      $resposta = [
          'success' => true,
          'message' => 'Executado com sucesso',
          'data' => [
              'cargos' => $cargo
          ]
      ];
      $response->getBody()->write(json_encode($resposta));
      return $response
          ->withHeader('Content-Type', 'application/json')
          ->withStatus(200);
  }
```

Controller: updateController

```php
public function updateController(Request $request, Response $response, array $args): Response {
        $idCargo = (int) $args['idCargo'];
        $body = $request->getBody()->getContents();
        $objPHP = json_decode($body);
        $nomeCargo = $objPHP->cargo->nomeCargo;
        $this->cargoService->updateService($idCargo, $nomeCargo);
        $resposta = [
            'success' => true,
            'message' => 'Atualizado com sucesso',
            'data' => [
                'cargos' => [
                    [
                        'idCargo' => $idCargo,
                        'nomeCargo' => $nomeCargo
                    ]
                ]
            ]
        ];
        $response->getBody()->write(json_encode($resposta));
        return $response
            ->withHeader('Content-Type', 'application/json')
            ->withStatus(200);
    }
```

Controller: deleteController

```php
public function deleteController(Request $request, Response $response, array $args): Response {
    $idCargo = (int) $args['idCargo'];
    $this->cargoService->deleteService($idCargo);
    $resposta = [
        'success' => true,
        'message' => 'Excluído com sucesso',
        'data' => [
            'cargos' => [
                [
                    'idCargo' => $idCargo
                ]
            ]
        ]
    ];

    $response->getBody()->write(json_encode($resposta));
    return $response
        ->withHeader('Content-Type', 'application/json')
        ->withStatus(200);
}
```

Controller: deleteController

```php
    public function countController(Request $request, Response $response, array $args): Response    {
        $total = $this->cargoService->countService();
        $resposta = [
            'success' => true,
            'message' => 'Executado com sucesso',
            'data' => [
                'count' => $total
            ]
        ];
        $response->getBody()->write(json_encode($resposta));
        return $response
            ->withHeader('Content-Type', 'application/json')
            ->withStatus(200);
    }
```

Service

A camada **Service** contém as regras de negócio mais complexas da aplicação.

Funções principais:

- desacoplar a lógica de negócio do Controller
- centralizar regras que envolvem múltiplas entidades
- permitir reutilização da lógica em diferentes requisições

O Service **não depende da camada web**, ou seja, ele não precisa saber se está sendo usado em:

- uma API
- um sistema MVC
- outro tipo de aplicação

Service

O service é responsável pela regra de negócio. Ele recebe os dados do controller, valida e aplica regras, e só então chama o DAO para acessar o banco.

CargoService (Camada de Serviço)

- Responsável pela **lógica de negócio** da entidade Cargo
- Atua entre:
  - Controller (entrada)
  - DAO (acesso ao banco)
- Implementa o padrão de **arquitetura em camadas**

Função principal:

- Aplicar regras de negócio
- Validar dados
- Controlar operações antes de acessar o banco

Passos:

1. Recebe dados do Controller
2. Cria/manipula entidade `Cargo`
3. Aplica regras (ex: evitar duplicidade)
4. Chama o `CargoDAO`
5. Retorna resultado (ou lança erro)

`Request → Router → Middleware → Controller → Service → DAO(model) → Response`

```php
<?php
namespace Api\Services;

// Importações das classes necessárias
use Api\Models\Cargo;               // Model/Entidade Cargo (representa a tabela no banco)
use Api\DAO\CargoDAO;               // Data Access Object - Responsável pela comunicação com o banco
use Api\Http\ErrorResponse;          // Classe para respostas de erro personalizadas
use InvalidArgumentException;        // Exceção para argumentos inválidos

class CargoService{
    private CargoDAO $cargoDAO;

    public function __construct(CargoDAO $cargoDAODependency) {
        // Armazena a dependência recebida na propriedade privada
        $this->cargoDAO = $cargoDAODependency; // injeção de dependência
    }
```

CargoService: createService

```
    public function createService(stdClass $objPHP): Cargo {

        $cargo = new Cargo();
        $cargo->setNomeCargo($objPHP->cargo->nomeCargo);
        $resultado = $this->cargoDAO->findByField(
            'nomeCargo',
            $cargo->getNomeCargo()
        );

        if (count($resultado) > 0) {
            throw new ErrorResponse(
                400,
                "Cargo já existe",
                [
                    "message" =>
                    "O cargo {$cargo->getNomeCargo()} já existe"
                ]
            );
        }


        return $this->cargoDAO->create($cargo);
    }
```

CargoService: countService

```php
    public function countService(): int    {
        return $this->cargoDAO->count();
    }
```

CargoService: findAllService

```php
   public function findAllService(): array
    {
        return $this->cargoDAO->findAll();
    }
```

CargoService: findByIdService

```php
    public function findByIdService(int $idCargo): ?Cargo {
        $cargo = new Cargo();
        $cargo->setIdCargo($idCargo);
        return $this->cargoDAO->findById($cargo->getIdCargo());
    }
```

CargoService: updateService

```php
public function updateService(int $idCargo, string $nomeCargo): bool{

        // Cria objeto com os dados atualizados
        $cargo = new Cargo();
        $cargo->setIdCargo($idCargo);         // ID do cargo a ser atualizado
        $cargo->setNomeCargo($nomeCargo);      // Novo nome

        // Delega a atualização para o DAO (UPDATE cargos SET ... WHERE idCargo = ?)
        return $this->cargoDAO->update($cargo);
    }
```

CargoService: updateService

```php
    public function deleteService(int $idCargo): bool{

        // Cria objeto com o ID
        $cargo = new Cargo();
        $cargo->setIdCargo($idCargo);

        // Delega a exclusão para o DAO (DELETE FROM cargos WHERE idCargo = ?)
        return $this->cargoDAO->delete($cargo);
    }
```

DAO (Data Access Object)

O **DAO** é um padrão de projeto que abstrai o acesso aos dados de uma aplicação.

Ele cria uma camada entre o **Model** e o **banco de dados**, evitando que a lógica de negócio conheça detalhes de implementação como:

- SQL, queries complexas
- Estrutura de acesso não relacional

DAO

O DAO é responsável por toda comunicação com o banco. Ele recebe dados do service, executa SQL e retorna o resultado, sem conter regras de negócio.

`Request → Router → Middleware → Controller → Service → DAO(model) → Response`

- **CargoDAO (Data Access Object)**
  - •Responsável por acessar o **banco de dados**
  - •Executa operações SQL (INSERT, SELECT, UPDATE, DELETE)
- **Função principal:**
  - Persistir dados
  - Buscar informações no banco
  - Encapsular o SQL da aplicação

Passos:

1. Recebe dados do Service (objeto `Cargo`)
2. Monta e executa SQL (via `MysqlDatabase`)
3. Retorna resultado:
   1. ID criado
   2. dados encontrados
   3. sucesso/erro

```php
<?php

namespace Api\DAO;

use Api\Models\Cargo;
use Api\Database\MysqlDatabase;
use Exception;

class CargoDAO  {
    private MysqlDatabase $database;
    public function __construct(MysqlDatabase $databaseInstance)    {
        $this->database = $databaseInstance;

  }
```

DAO: create

```php
public function create(Cargo $objCargo): Cargo {
        $sql = "
            INSERT INTO cargo (nomeCargo)
            VALUES (:nomeCargo)
        ";
        $parametros = [
            ':nomeCargo' => $objCargo->getNomeCargo()
        ];
        $stmt = $this->database->getConnection()->prepare($sql);

        if (!$stmt->execute($parametros)) {
            throw new Exception("Erro ao cadastrar cargo.");
        }
        $novoID = (int) $this->database->getConnection()->lastInsertId();
        $objCargo->setIdCargo($novoID);
        return $objCargo;
    }
```

DAO: delete

```php
public function delete(Cargo $objCargoModel): bool {
        $sql = "
            DELETE FROM cargo
            WHERE idCargo = :idCargo
        ";
        $parametros = [
            ':idCargo' => $objCargoModel->getIdCargo()
        ];

        $stmt = $this->database->getConnection()->prepare($sql);
        $stmt->execute($parametros);


       return $stmt->rowCount() > 0;
    }
```

DAO: update

```php
public function update(Cargo $objCargoModel): bool {
        $sql = "
            UPDATE cargo
            SET nomeCargo = :nomeCargo
            WHERE idCargo = :idCargo
        ";
        $parametros = [
            ':nomeCargo' => $objCargoModel->getNomeCargo(),
            ':idCargo' => $objCargoModel->getIdCargo()
        ];
        $stmt = $this->database->getConnection()->prepare($sql);
        $stmt->execute($parametros);
        return $stmt->rowCount() > 0;
    }
```

DAO: findAll

```php
public function findAll(): array {
        $sql = "SELECT * FROM cargo";
        $stmt = $this->database->getConnection()->query($sql);
        $matrizArrays = $stmt->fetchAll(\PDO::FETCH_ASSOC);
        $cargos = [];
        foreach ($matrizArrays as $linhaMatriz) {
            $cargo = new Cargo();


            $cargo->setIdCargo((int) $linhaMatriz['idCargo']);
            $cargo->setNomeCargo($linhaMatriz['nomeCargo']);


            $cargos[] = $cargo;
        }
        return $cargos;
    }
```

DAO: count

```php
    public function count(): int {
        $sql = "SELECT COUNT(*) AS qtd FROM cargo";
        $stmt = $this->database->getConnection()->query($sql);
        $linhaMatriz = $stmt->fetch(\PDO::FETCH_ASSOC);
        return (int) $linhaMatriz['qtd'];
     }
```

DAO: findById

```php
    public function findById(int $idCargo): ?Cargo {
        $resultado = $this->findByField('idCargo', $idCargo);
        if (!empty($resultado)) {
            return $resultado[0];
        }
        return null;
    }
```

DAO: findById

```php
public function findByField(string $field, $value): array {
        $camposPermitidos = [
            'idCargo',
            'nomeCargo'
        ];
        if (!in_array($field, $camposPermitidos)) {
            throw new Exception("Campo inválido.");
        }
        $sql = "SELECT * FROM cargo WHERE $field = :value";
        $stmt = $this->database->getConnection()->prepare($sql);
        $stmt->execute([
            ':value' => $value
        ]);
        $matrizArrays = $stmt->fetchAll(\PDO::FETCH_ASSOC);
        $cargos = [];
        foreach ($matrizArrays as $linhaMatriz) {
            $cargo = new Cargo();
            $cargo->setIdCargo((int) $linhaMatriz['idCargo']);
            $cargo->setNomeCargo($linhaMatriz['nomeCargo']);
            $cargos[] = $cargo;
        }
        return $cargos;
    }
```

Model

O **Model** é responsável por:

- definir a estrutura dos dados
- representar entidades do sistema
- definir tipos de campos e valores padrões
- Também pode conter **regras de negócio do domínio**, ou seja, regras relacionadas apenas à própria entidade.

O Model pode realizar verificações como:

- tipos de dados
- consistência de atributos
- validações simples de dados
  - Exemplo:
    - nome com no mínimo 5 caracteres
    - Validar cpf
    - Validar idade menor que 120 anos.

Model

O Model representa a entidade do sistema, garantindo a integridade dos dados através de validações e permitindo sua conversão para JSON.

Service/DAO → Cargo → JSON

`Request → Router → Middleware → Controller → Service → DAO(model) → Response`

Cargo (Model / Entidade)

- Representa os **dados da tabela Cargo** no sistema
- Contém os atributos:
  - `idCargo`
  - `nomeCargo`
- Implementa `JsonSerializable`

Função principal:

- Armazenar dados da entidade
- Validar informações básicas
- Converter objeto para JSON

Passos:

1. Recebe valores pelos setters (`setIdCargo`, `setNomeCargo`)
2. Valida os dados (ex: não vazio, tamanho mínimo)
3. Se inválido → lança `InvalidArgumentException`
4. Se válido → armazena no objeto
5. Pode ser convertido para JSON (`jsonSerialize()`)

Lembre-se

- json_encode transforma um vetor em string json.

```php
<?php
namespace Api\Models;
use InvalidArgumentException;
use \JsonSerializable;
class Cargo implements JsonSerializable {
    private int $idCargo;
    private string $nomeCargo = "";
    public function __construct(){}
    public function getIdCargo(): ?int    {
        return $this->idCargo;
    }
    public function setIdCargo(int $value): void  {
        if (!is_int($value)) {
            throw new InvalidArgumentException("idCargo deve ser um número inteiro.");
        }
        if ($value <= 0) {
            throw new InvalidArgumentException("idCargo deve ser maior que zero.");
        }
        $this->idCargo = $value;
    }
    public function getNomeCargo(): ?string {
        return $this->nomeCargo;
    }
```

```php
    public function setNomeCargo(string $value): void     {
        $nome = trim($value);
        if ($nome === '') {
            throw new InvalidArgumentException("nomeCargo não pode ser vazio.");
        }
        $len = mb_strlen($nome);

        if ($len < 3) {
            throw new InvalidArgumentException("nomeCargo deve ter pelo menos 3 caracteres.");
        }
        if ($len > 64) {
            throw new InvalidArgumentException("nomeCargo deve ter no máximo 64 caracteres.");
        }
        $this->nomeCargo = $nome;
    }
    public function jsonSerialize(): array
    {
        return [
            'idCargo' => $this->getIdCargo(),
            'nomeCargo' => $this->getNomeCargo()
        ];
    }
}
```

json_encode transforma um vetor em string json.

json_encode transforma um vetor em string json.

- $c = new Cargo();
- $c->setNomeCargo(“Professor”);
- $c->setIdCargo(1);
- $stringJson = json_encode($c);

Ao json_encode em uma classe com jsonSerialize o método é chamado e retorna um vetor associativo

```php
public function jsonSerialize(): array
{
    return [
        'idCargo' => $this->getIdCargo(),
        'nomeCargo' => $this->getNomeCargo()
    ];
}
```

Utilizando o insomnia

- Crie a base de dados a partir do arquivo:
  - Src/docs/banco.sql
- Rode a aplicação no terminal:
  - c:\xampp\php\php.exe -S localhost:8080 -t Public/

GET http://localhost:8080/cargos Send

200 OK 266 ms 437 B 1 Hour Ago

Params Body Auth Headers 3 Scripts Docs

Preview Headers 8 Cookies Tests 0 / 0 → Mock Console

No Body

Preview

Enter a URL and send to get a response

Select a body type from above to send data in the body of a request


```
{
  "success": true,
  "message": "Busca realizada com sucesso",
  "data": {
    "cargos": [
      {
        "idCargo": 1,
        "nomeCargo": "Administrador"
      },
      {
        "idCargo": 9,
        "nomeCargo": "Administrador 007"
      },
      {
        "idCargo": 8,
        "nomeCargo": "Administrador 008"
      },
      {
        "idCargo": 7,
        "nomeCargo": "Administrador 2"
      },
      {
```