

MVCS

(Model-View-Controller-Service)

Prof. Me. Hélio Esperidião

Regra de negócio



Uma regra de negócio é uma diretriz ou condição que define ou restringe como um processo dentro de uma organização deve funcionar.



Em outras palavras, é um conjunto de instruções ou critérios que orientam o comportamento da empresa, apoiam a tomada de decisão e garantem que os processos sigam os objetivos estratégicos do negócio.



Uma **regra de negócio** é uma condição, restrição ou cálculo que define **como o sistema deve se comportar para atender às necessidades da empresa.**

Exemplos de regras de negócio:

E-commerce: “Não permitir finalizar a compra se o carrinho estiver vazio.”

Banco: “A conta corrente não pode ter saldo negativo, exceto se o cliente tiver limite de cheque especial.”

Sistema de RH: “Um funcionário só pode registrar horas extras se tiver autorização do gestor.”

Introdução ao MVCS

MVCS é uma evolução do padrão MVC.

Adiciona a camada **Service** para **organizar regras de negócio complexas**.

Usado em aplicações modernas para maior modularidade e manutenção.

Componentes da Arquitetura

Model:

- Representa os dados, regras de armazenamento e manutenção de dados e regras de domínio.

View:

- interface com o usuário (UI).

Controller:

- intermediário entre View e Service.

Service:

- Concentra lógica de negócio complexa.

Model

Define estruturas de dados e persistência.

- Tipos dos campos, valores padrões, etc.

Regra de negócio do **domínio**, ou seja tem relação apenas com a entidade.

O modelo pode fazer verificações dos tipos de dados da entidade referida.

Validações no model?

Garantir integridade dos dados

Reduzir erros mais cedo

Facilitar manutenção e testes

Maior segurança contra inconsistência.

Colocar validações no *model* garante que **seu objeto nunca esteja em um estado inválido**. Isso é um dos princípios de **boas práticas de Programação orientada a objetos**.

Exemplo de model

```
module.exports = class Cargo {  
  // Atributos privados  
  #idCargo;  
  #nomeCargo;  
  get idCargo() {  
    return this.#idCargo;  
  }  
  
  set idCargo(value) {  
    const parsed = Number(value); // Converte o valor para número  
    if (!Number.isInteger(parsed)) { // Verifica se é um número inteiro  
      throw new Error("idCargo deve ser um número inteiro.");  
    }  
    if (parsed <= 0) { // Verifica se é maior que zero  
      throw new Error("idCargo deve ser maior que zero.");  
    }  
    this.#idCargo = parsed; // Atribui valor ao atributo privado  
  }  
  
  get nomeCargo() {  
    return this.#nomeCargo;  
  }  
  
  set nomeCargo(value) {  
    if (typeof value !== "string") { // Verifica se é string  
      throw new Error("nomeCargo deve ser uma string.");  
    }  
    // Remove espaços no início/fim  
    const nome = value.trim();  
    // Verifica comprimento mínimo  
    if (nome.length < 3) {  
      throw new Error("nomeCargo deve ter pelo menos 3 caracteres.");  
    }  
    // Atribui valor ao atributo privado  
    this.#nomeCargo = nome;  
  }  
}
```

View



Responsável pela
apresentação visual.



Mostra os dados
processados ao usuário.



Exemplo: páginas web, telas
mobile, relatórios.

Controller

Recebe requisições da View.

Chama os Services adequados.

Pertence à camada de apresentação (MVC).

Responsável por receber a requisição do usuário, chamar o domínio/serviços adequados e retornar a resposta.

Não deve conter regras de negócio.

Foco: orquestração de entrada/saída.

Retorna dados para a View.

Control

```
store = async (request, response, next) => {
  console.log("● CargoControle.store()");
  try {
    const cargoBodyRequest = request.body.cargo;
    const novoId = await this.#cargoService.createCargo(cargoBodyRequest);
    const objResposta = {
      success: true,
      message: "Cadastro realizado com sucesso",
      data: {
        cargos: [{
          idCargo: novoId,
          nomeCargo: cargoBodyRequest.nomeCargo
        }]
      }
    };
    if (novoId) {
      response.status(200).send(objResposta);
    } else {
      throw new Error("Falha ao cadastrar novo Cargo");
    }
  } catch (error) {
    next(error); // Encaminha o erro para o middleware de tratamento
  }
}
```

Service

Contém regras de negócio complexas (que não cabem em uma única entidade).

Desacopla a lógica de negócio do Controller e do Middleware.

Retira do Controller a responsabilidade de validar regras de negócio.

Pode ser reutilizado em diferentes Controllers.

É independente da camada web:

- Não sabe se está sendo usado numa API, MVC, ou outro tipo de app.

Exemplos:

- Cálculo de impostos.
- Processar uma transferência entre contas.
- Regras de validação entre várias entidades.

Service

```
createCargo = async (cargoBodyRequest) => {  
  console.log("🟡 CargoService.createCargo()");
```

```
    const cargo = new Cargo();
```

```
    cargo.nomeCargo = cargoBodyRequest.nomeCargo; //regra de domínio chamada
```

```
    //regra de negocio
```

```
    const resultado = await this.#cargoDAO.findByField("nomeCargo", cargo.nomeCargo);
```

```
    if (resultado.length > 0) {  
      throw new ErrorResponse(  
        400,  
        "Cargo já existe",  
        { message: `0 cargo ${cargo.nomeCargo} já existe` }  
      );  
    }
```

```
    return this.#cargoDAO.create(cargo);
```

```
  }
```

DAO



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



Ele cria uma camada entre o domínio (Model) e a base de dados, de forma que a lógica de negócio não precise conhecer detalhes do banco (SQL, ORM, queries complexas).

Objetivos do DAO

Separar responsabilidades:	o Model ou Service não faz queries diretamente.
Facilitar manutenção:	se mudar o banco, só o DAO precisa ser atualizado.
Reutilização:	várias partes da aplicação podem usar o mesmo DAO.
Isolamento:	ajuda em testes unitários.

Estrutura típica

Entidade
(Model) :

- representa os dados e regras do domínio.

DAO:

- fornece métodos como Salvar(), Atualizar(), Excluir(), ObterPorId().

Service:

- usa o DAO para manipular dados sem saber como eles são armazenados.

DAO

```
create = async (objCargo) => {  
    const SQL = "INSERT INTO cargo (nomeCargo) VALUES (?);";  
    const params = [objCargo.nomeCargo];  
  
    const [resultado] = await this.#database.execute(SQL, params);  
  
    if (!resultado.insertId) {  
        throw new Error("Falha ao inserir cargo");  
    }  
  
    console.log("✅ CargoDAO.create()");  
    return resultado.insertId;  
};
```

MVCS + DAO + Middleware + Roteador



View: interação do usuário.



Roteador: direciona a requisição ao Controller correto.



Middleware: validações e processos transversais (autenticação, logging).



Controller: orquestra entrada/saída. Service: lógica de negócio complexa.



Model: entidades + regras do domínio.

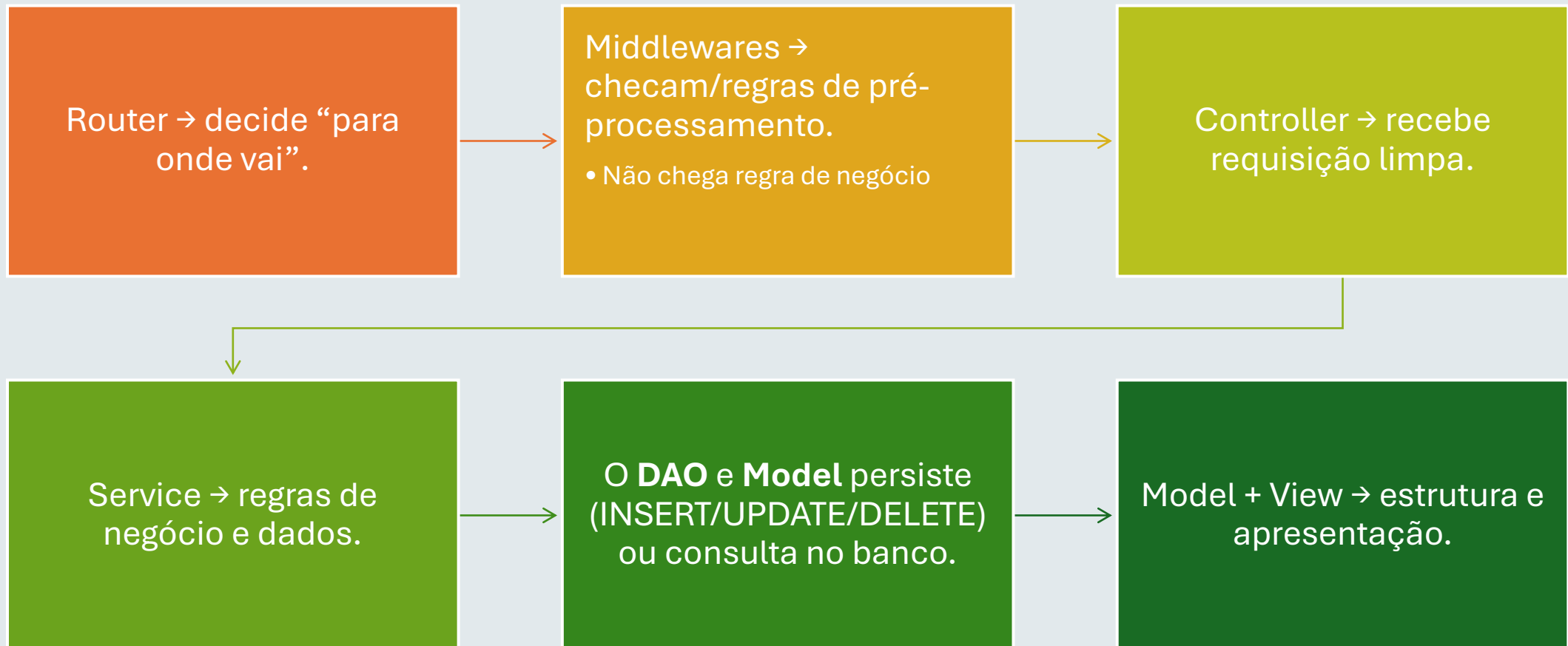


DAO / Repositório: abstrai a persistência e acesso a dados.

Detalhes: Fluxo de Dados no MVCS

1. Usuário interage com a View
 1. Envia dados via formulário, botão ou API call.
2. Roteador (Router) recebe a requisição
 1. Encaminha a requisição para o Controller correto.
3. Middleware
 1. Pode executar funções transversais como autenticação, autorização, logging ou validação genérica.
4. Controller recebe a requisição
 1. Orquestra a execução, chama o Service adequado.
5. Service executa a lógica de negócio
 1. Manipula o Model e coordena processos complexos.
6. Model interage com DAO / Repositório
 1. DAO abstrai a persistência (CRUD, consultas, transações).
 2. Service chama DAO para gravar, atualizar ou ler dados.
7. Resultado retorna ao Service => Controller
8. Controller converte para o formato apropriado (JSON, HTML, etc.).
9. Controller envia resposta para a View
 1. Usuário recebe o resultado final.

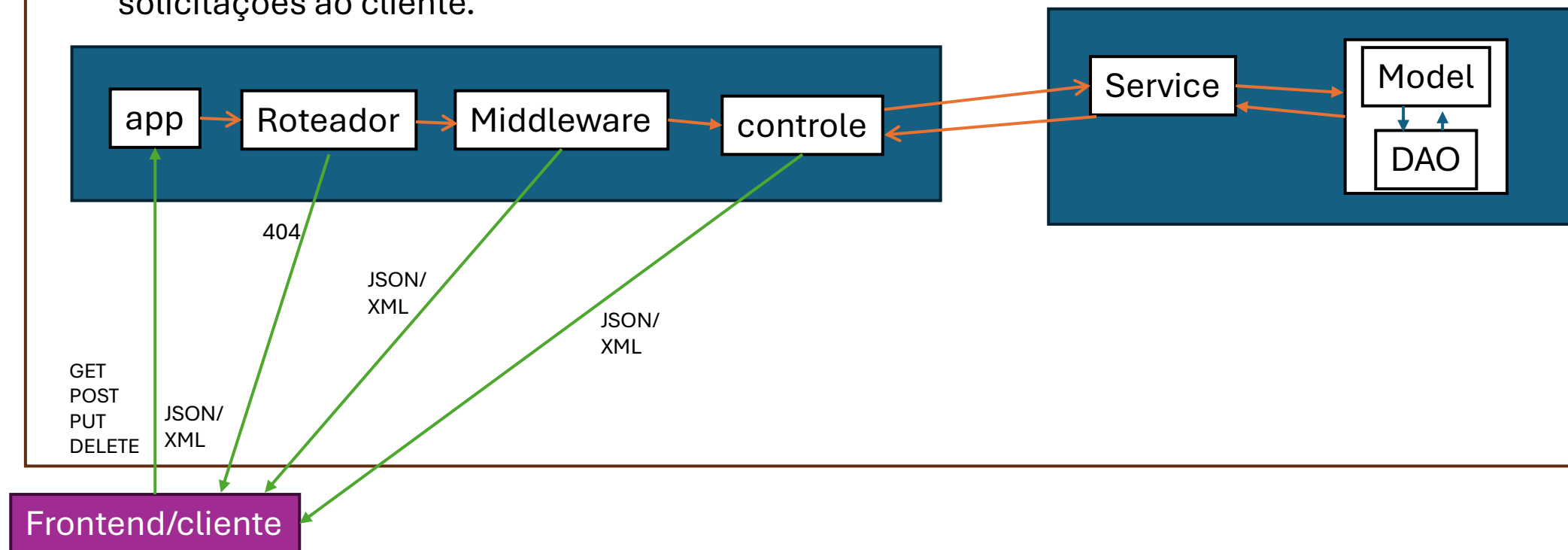
Em resumo:



api

Conhece o requisitante e pode responder solicitações ao cliente.

Não Conhece o requisitante, o requisitante pode ser trocado



O que é Injeção de Dependência (Dependency Injection – DI)

- Injeção de dependência é um padrão de projeto usado para reduzir acoplamento entre classes, permitindo que uma classe receba suas dependências de fora em vez de criá-las internamente.
- Ou seja: Em vez da classe criar seus objetos necessários, alguém “injeita” esses objetos nela.
- Isso facilita testes, manutenção e reutilização.
- Benefícios:
 - Desacoplamento: a classe não depende diretamente de implementações concretas.
 - Facilidade de testes: podemos injetar mocks ou stubs no lugar das dependências reais.
 - Reuso e flexibilidade: a mesma classe pode ser usada com diferentes implementações de dependências.
 - Manutenção simplificada: mudanças em dependências não exigem alterar a classe que as usa.

Server.js

```
setupCargo = () => {
  console.log("⬆ Server.setupCargo()");
  // ♦ Middleware de validação para Cargo
  // Verifica se os dados recebidos nas requisições estão corretos
  // antes de passar para o Controller. Isso mantém a lógica de validação
  // separada da lógica de negócio.
  this.#cargoMiddleware = new CargoMiddleware();
  // ♦ DAO de Cargo: Recebe a conexão com o banco (pool MySQLDatabase) via injeção de dependência.
  this.#cargoDAO = new CargoDAO(this.#database);
  // ♦ Service de Cargo: Recebe o DAO via injeção de dependência.
  // O Service contém a lógica de negócio da entidade Cargo
  this.#cargoService = new CargoService(this.#cargoDAO);
  // ♦ Controller de Cargo: Recebe o Service via injeção de dependência.
  this.#cargoControl = new CargoControl(this.#cargoService);
  // ♦ Roteador de Cargo
  // Recebe todas as dependências necessárias:
  // - jwtMiddleware → autenticação
  // - cargoMiddleware → validação de entrada
  // - cargoControl → manipulação da lógica de negócio
  const cargoRoteador = new CargoRoteador(
    this.#jwtMiddleware,
    this.#cargoMiddleware,
    this.#cargoControl
  );
  // Todas as rotas da entidade Cargo ficam disponíveis em:
  // http://localhost:PORT/api/v1/cargos
  this.#app.use("/api/v1/cargos", cargoRoteador.createRoutes());
}
```

Vantagens do MVCS

Maior organização do código.

Separação de responsabilidades clara.

Facilita testes unitários.

Escalabilidade em aplicações grandes.

MVCS melhora a manutenibilidade em relação ao MVC.

Torna aplicações mais modulares.

Muito utilizado em frameworks modernos:

- (Spring, Angular, Java / Spring Framework, Laravel (PHP))