Arquitetura Web: View, Router, middleware, Controller, Service, DAO e Model

Prof. Me. Hélio Lourenço Esperidião Ferreira

Entidades do sistema

São os **objetos do domínio**, ou seja, as coisas importantes para o negócio:

- Cliente
- Pedido
- Produto
- Conta

Elas representam **conceitos do mundo real dentro do sistema**

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 critérios ou instruções que orienta o comportamento do sistema, apoia a tomada de decisões e garante que os processos atendam aos objetivos do negócio.

Uma regra de negócio pode ser:

- uma condição
- uma restrição
- um cálculo

Define **COMO O SISTEMA deve se comportar para atender às necessidades da empresa**.

Exemplos de regras de negócio

1. Condição

- Define quando algo pode ou não acontecer.
  - “Pedidos acima de R$ 200 têm frete grátis.”
  - “Um usuário só pode avaliar um produto após comprá-lo.”

Restrição

- Impõe limites ou regras obrigatórias.
  - “Um CPF só pode ser cadastrado uma vez no sistema.”
  - “O limite máximo de parcelas é 12.”
  - “Não é permitido cancelar pedidos já enviados.”

Exemplos de regras de negócio

Cálculo

- Define como valores devem ser calculados.
  - “O valor total do pedido = soma dos produtos + frete – desconto.”
  - “Desconto de 10% para clientes VIP.”
  - “Juros de 2% ao mês em pagamentos atrasados.”

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.”

Regra de Domínio

Elas definem **como as entidades do sistema devem se comportar**, garantindo que os dados e operações respeitem as regras do negócio.

Essas regras geralmente estão relacionadas **diretamente às entidades do sistema**.

Características das regras de domínio

- Estão relacionadas às **entidades do sistema**
- Garantem **consistência e integridade dos dados**

Exemplos de Regras de domínio

O preço do produto deve ser maior que zero

CPF deve ser válido

Email deve ser válido

Nome mais do que 10 caracteres

INVARIANTE

Uma invariante é uma regra que **nunca pode ser violada durante a vida do objeto**

Exemplo no Quadrado

lado > 0

Se essa regra for quebrada, o objeto deixa de representar um quadrado válido.

Relação entre Regra de Domínio e Invariante

Regra de domínio

- Elas definem **como as entidades do sistema devem se comportar**, garantindo que os dados e operações respeitem as regras do negócio.

Invariante

- É uma **condição que deve ser sempre verdadeira para um objeto válido** durante todo o seu ciclo de vida.

Ou seja:

- As **invariantes ajudam a garantir que as regras de domínio sejam respeitadas dentro das entidades do sistema.**

Sistema bancário

Regra de domínio

- Uma conta não pode ter saldo negativo sem limite de crédito.

Invariante da classe Conta

- saldo ≥ -limite

Enquanto o objeto existir, **essa condição deve sempre ser verdadeira**.

Sistema de pedidos

- **Regra de domínio**
  - Um pedido deve possuir pelo menos um item.
- **Invariante da classe Pedido**
  - quantidadeDeItens > 0

Desmistificando

Regra de domínio:

- definem **como as entidades do sistema devem se comportar**

Invariante:

- É uma **regra de domínio que deve ser sempre verdadeira**, garantindo que a entidade nunca fique em um estado inválido

INVARIANTE

Uma invariante é uma regra que **nunca pode ser violada durante a vida do objeto**

Exemplo no Quadrado

lado > 0

Se essa regra for quebrada, o objeto deixa de representar um quadrado válido.

Exemplo com validação

- **O que está acontecendo aqui?**
  - O método recebe o valor.
  - Antes de alterar o estado do objeto, verifica a regra.
  - Se for inválido, lança uma exceção.
  - O objeto nunca entra em estado inconsistente.

```php
public function setLado(float $novoLado): void{
   if ($novoLado <= 0) {
      throw new \InvalidArgumentException(
           'O lado do quadrado não pode ser negativo.'
       );
   }
        $this->lado = $novoLado;
}
```

Arquitetura MVC Estendida (Router, Middleware, Service, DAO e Model)

O padrão **MVC (Model-View-Controller)** é amplamente utilizado para separar responsabilidades em sistemas.

Em aplicações modernas, ele é expandido para melhorar organização, escalabilidade e manutenção, incorporando novas camadas:

- **Router**
- **Middleware**
- **Service**
- **DAO (Data Access Object)**
- **Model**

Essa estrutura representa uma evolução prática do MVC tradicional, comum em APIs modernas.

Objetivo da Arquitetura

Separar claramente:

- Entrada da requisição
- Controle de fluxo
- Regras de negócio
- Acesso a dados
- Estrutura dos dados

Estrutura Geral

- Request
  - Router
  - Middleware
  - Controller
    - Recebe dados e encaminha para o service
    - Response (View)
  - Service
  - DAO
  - Model
  - Banco de Dados

Camadas da Arquitetura

1. Router

Responsabilidades:

- Definir as rotas da aplicação
- Mapear URLs para Controllers
- Direcionar requisições corretamente

Características:

- Camada inicial da aplicação
- Não contém lógica de negócio
- Pode associar middlewares às rotas

Exemplo conceitual:

- “POST /pedidos → Controller de Pedido”

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**.

Endpoints de um roteador

GET /usuarios

- Controller.listar()

POST /usuarios

- Controller.criar()

GET /usuarios/:id

- Controller.buscar()

DELETE /usuarios/:id

- Controller.remover()

2. Middleware

Responsabilidades:

- Interceptar requisições antes do Controller
- Executar regras transversais
- Validar, autenticar e autorizar

Exemplos:

- Autenticação (usuário logado)
- Autorização (permissões)
- Validação de dados
- Logs e auditoria

Características:

- Reutilizável
- Independente de regra de negócio
- Pode bloquear a requisição

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 (restringe quem pode acessar a api... Quais domínios, quais IPs..)

Validação de dados no middleware

A **validação da estrutura dos dados** pode ser realizada no **Middleware**, antes que a requisição chegue ao Controller.

Isso garante que **dados inválidos sejam interceptados antecipadamente**, evitando processamento desnecessário na aplicação.

Objetivos da validação no Middleware

- Verificar se os **dados obrigatórios foram enviados**
- Validar **formatos de dados** (e-mail, datas, números)
- Evitar que dados inválidos cheguem ao **Controller e ao Service**
- Melhorar a **segurança e consistência da aplicação**
- Essa camada não valida conteúdo, valida apenas estrutura

Middlewares

Vantagens

- Evita processamento desnecessário no Controller
- Melhora a organização do código
- Aumenta a segurança da aplicação

O Middleware funciona como um **porteiro da aplicação**, antes de permitir que ela continue no fluxo.

3. Controller

Responsabilidades:

- Receber requisições já tratadas
- Validar entrada básica
- Chamar o Service
- Retornar resposta

Características:

- Deve ser simples (thin controller)
- Não contém regras de negócio
- Atua como intermediário

Controller

O **Controller** é responsável por:

- receber a requisição **encaminhada pelo Router (ou middleware se houver)**
- chamar o Service necessário
- 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**

4. Service

Responsabilidades:

- Implementar regras de negócio
- Orquestrar operações
- Definir fluxo da aplicação

Aqui ficam:

- Casos de uso
- Processos do sistema
- Decisões do negócio

Características:

- Camada central da lógica
- Independente de HTTP ou interface

Service

A camada **Service** contém as regras 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

Exemplos de Services

- cálculo de frete e prazo de entrega
- aplicação de descontos e cupons
- verificação de saldo antes de uma operação
- conciliação de pagamentos
- cálculo de juros e multas
- validação de regras de negócio complexas (ex: limite de crédito)
- agregação de dados de múltiplos repositórios
- integração com APIs externas (ex: gateway de pagamento)
- cálculo de comissões

5. DAO (Data Access Object)

Responsabilidades:

- Isolar o acesso ao banco de dados
- Executar operações de persistência
- Abstrair consultas

Exemplos de operações:

- Salvar dados
- Buscar registros
- Atualizar informações
- Remover dados

Características:

- Não contém regra de negócio
- Facilita troca de banco (MongoDB, SQL, etc.)
- Centraliza queries

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 aos dados

Objetivos do DAO

Separação de responsabilidades

- Model ou Service não executam queries (consultas ao banco) diretamente.

Facilidade de manutenção

- se o banco de dados mudar, apenas o DAO precisa ser atualizado.

Reutilização

- várias partes da aplicação podem usar o mesmo DAO.

Isolamento

- facilita a criação de testes unitários.

6. Model

Responsabilidades:

- Representar entidades do sistema
- Definir estrutura dos dados
- Aplicar regras simples (invariantes básicas)

Exemplos:

- Tipos de dados
- Campos obrigatórios
- Restrições simples

Características:

- Ligado ao banco
- Base da persistência
- Não deve conter lógica complexa

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 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.

Validações no Model

Colocar validações no Model ajuda a:

- garantir a integridade dos dados
- reduzir erros
- facilitar manutenção e testes
- aumentar a segurança contra inconsistências

Isso garante que o objeto **nunca esteja em um estado inválido**, o que é uma boa prática da **Programação Orientada a Objetos**.

7. View (Response)

Responsabilidades:

- Retornar dados ao cliente
- Formatar resposta (JSON, HTML, etc.)

Características:

- Apenas apresentação
- Sem lógica de negócio

Camadas da aplicação

View	interface que o usuário vê.
Router	decide qual endpoint será chamado.
Middleware	Ele funciona como um **filtro ou interceptador** da requisição.
Controller	recebe a requisição e coordena a ação.
Service	contém a lógica de negócio.
DAO	responsável por acessar o banco de dados.
Model	representa a estrutura dos dados (entidades/tabelas). Possui as regras de domínio
Resposta volta para a View.	Envia um json para o cliente.

Fluxo de Execução

- Cliente envia requisição
- Router identifica a rota
- Middleware intercepta e valida
- Controller recebe a requisição
- Controller chama o Service
- Service executa regras de negócio
- Service chama o DAO
- DAO acessa o Model
- Model interage com o banco
- Resultado retorna até o Controller
- Response é enviada ao cliente

Problemas sem essas camadas

Sem Router:

- Falta de organização nas rotas

Sem Middleware:

- Código duplicado
- Falta de segurança centralizada

Sem Service:

- Controllers complexos
- Regras espalhadas

Sem DAO:

- Acesso direto ao banco espalhado
- Forte acoplamento

Vantagens da Arquitetura

- Alta organização
- Separação de responsabilidades
- Facilidade de manutenção
- Reutilização de código
- Baixo acoplamento
- EscalabilidadeTestabilidade

Vantagens da Arquitetura

- Separação clara de responsabilidades
- Facilidade para testes unitários
- Maior escalabilidade em aplicações grandes
- Melhor manutenção do sistema
- A arquitetura melhora a manutenibilidade em relação ao **MVC**
- Torna as aplicações mais modulares.