



5954014 - Engenharia de Software

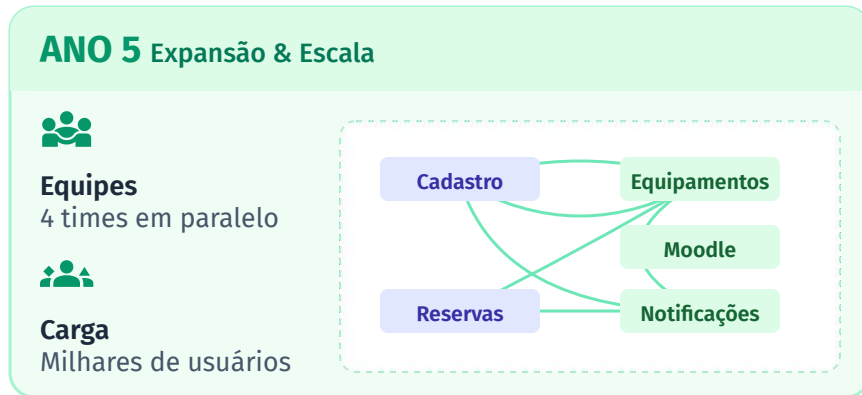
Aula 12 - Arquitetura de Software

Prof. Dr. Denis Martins

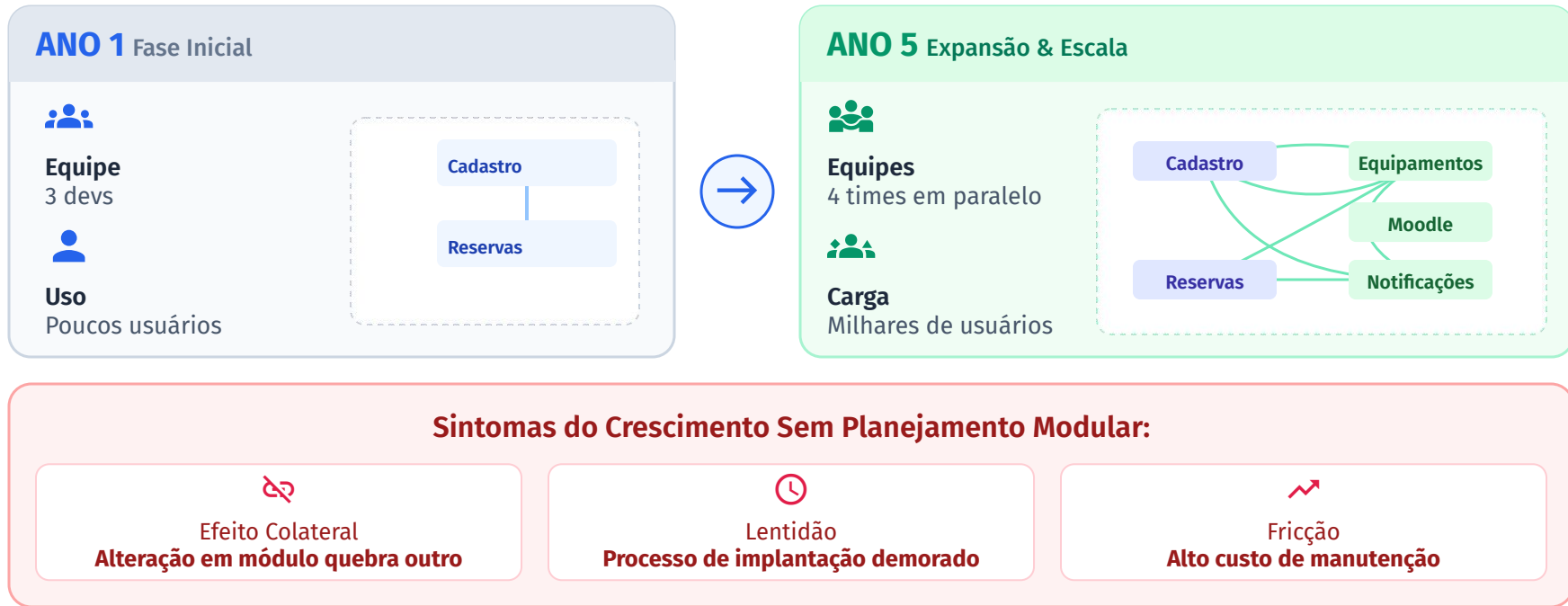
DCM | FFCLRP | USP

**Como organizar componentes e dependências
para criar sistemas que possam evoluir?**

Motivação: Evolução da Arquitetura



Motivação: Evolução da Arquitetura



Motivação: Evolução da Arquitetura



Sintomas do Crescimento Sem Planejamento Modular:



Efeito Colateral
Alteração em módulo quebra outro



Lentidão
Processo de implantação demorado



Fricção
Alto custo de manutenção

É apenas um problema de código ou também de arquitetura?

Objetivos de Aprendizagem



1. Design vs. Arquitetura

Diferenciar o escopo de Design e Arquitetura de Software.



2. Comportamento vs. Estrutura

Compreender a dicotomia entre Comportamento e Estrutura (Arquitetura Limpa).



3. Trade-offs de Sistemas

Avaliar os 10 principais Trade-offs de sistemas (escalabilidade, dados, comunicação).



4. Padrões Arquiteturais

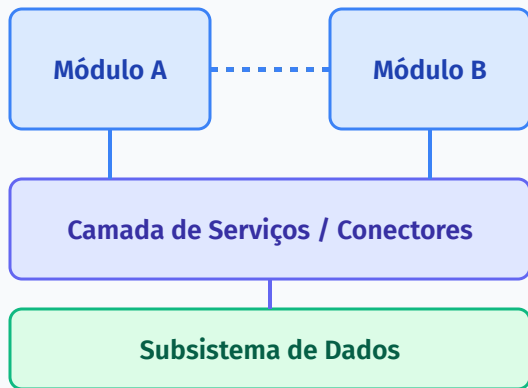
Mapear e aplicar Padrões (MVC, Microserviços, Hexagonal).

O que é Arquitetura de Software?

Arquitetura é a organização de alto nível dos elementos relevantes de um software, suas responsabilidades, interfaces e relações, associada às decisões estruturais que orientam sua evolução.

Projeto Arquitetural (Alto Nível)

Visão sistêmica, estrutura e limites



Projeto Detalhado (Baixo Nível / Zoom)

Estrutura interna, algoritmos e implementação

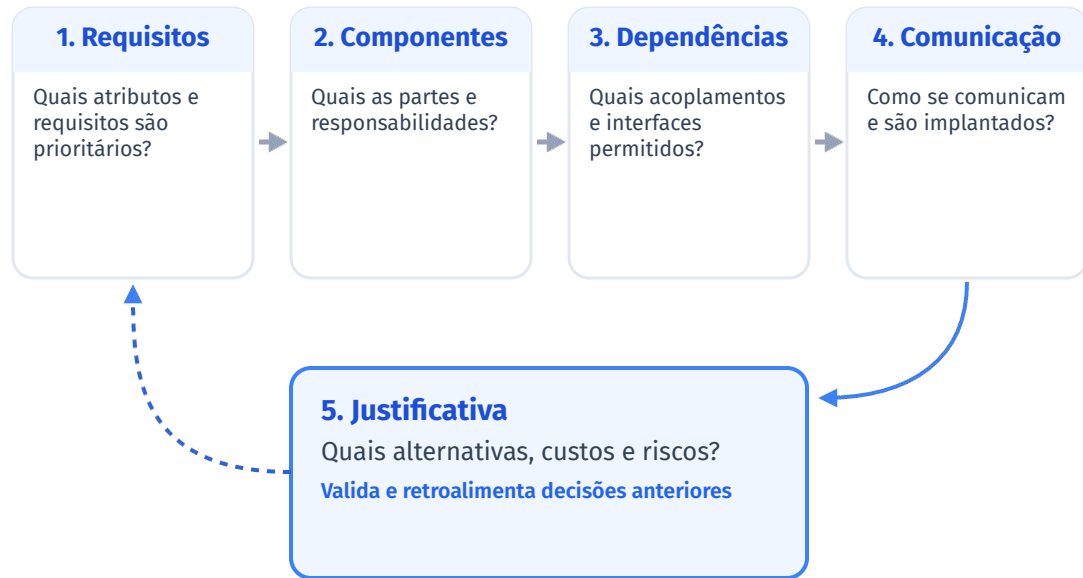
class UserProcessor

```
- userData: Map<String, Object>
- isValidated: Boolean

+ validateCredentials(): Boolean
+ processData(algo: Algorithm): Void
- internalHelper(): void
```

Toda arquitetura faz parte do projeto de software, mas nem toda decisão de projeto é arquitetural.

Arquitetura é uma Sequência de Decisões



Feedback Loop: Decisões de arquitetura passam por revisão contínua conforme novos requisitos e justificativas emergem.

Exemplo: Registro de Decisão (ADR)

DECISÃO

Separar envio de notificações por uma interface dedicada.

MOTIVAÇÃO

Facilitar a substituição ou adição de novos provedores (E-mail, SMS, Push).

ALTERNATIVAS

Acoplamento direto à SDK do provedor atual ou serviço de terceiros genérico.

CONSEQUÊNCIAS & CUSTO

Criação de nova camada de abstração (maior complexidade inicial de código).

Requisitos Não-Funcionais Orientam a Arquitetura



Manutenibilidade

Consigo trocar o provedor de e-mail sem reescrever reservas?

Microexemplo: Isolar a integração atrás de uma interface desacoplada (Camada de Abstração).



Desempenho

A consulta de salas responde em até 2 segundos?

Microexemplo: Aplicação de cache em memória (Redis) e otimização de queries de leitura.



Disponibilidade (Destaque)

Uma falha no envio de e-mails impede a reserva?

Microexemplo: Processamento assíncrono utilizando filas de mensagens (RabbitMQ/Kafka).



Testabilidade

Consigo testar conflito de horários sem banco de dados real?

Microexemplo: Injeção de dependências e uso de repositórios em memória para testes unitários.

“Uma escolha arquitetural favorece alguns atributos, mas pode introduzir custos em outros.”

Componentes, Interfaces, Conectores e Dependências

Componente

Unidade responsável por uma parte do comportamento.

Interface

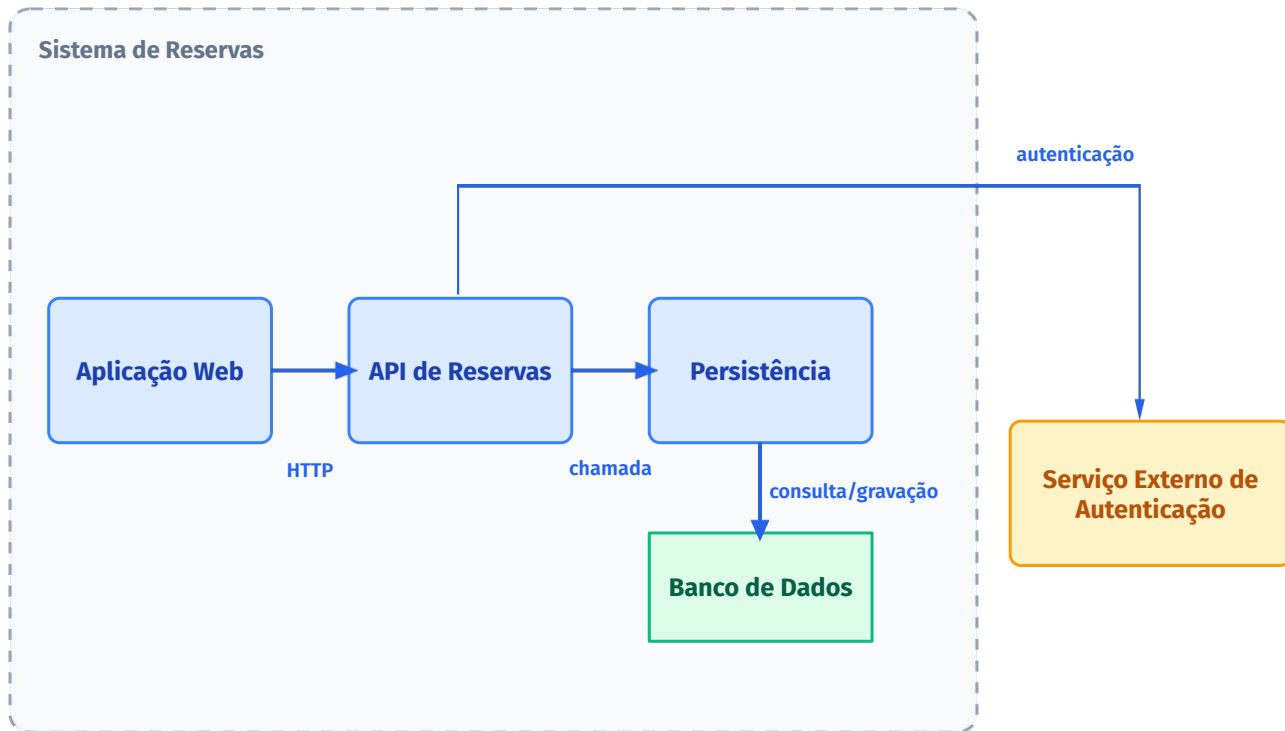
Contrato disponibilizado ou requerido.

Conector

Mecanismo de interação (chamada de API, mensagem, evento).

Dependência

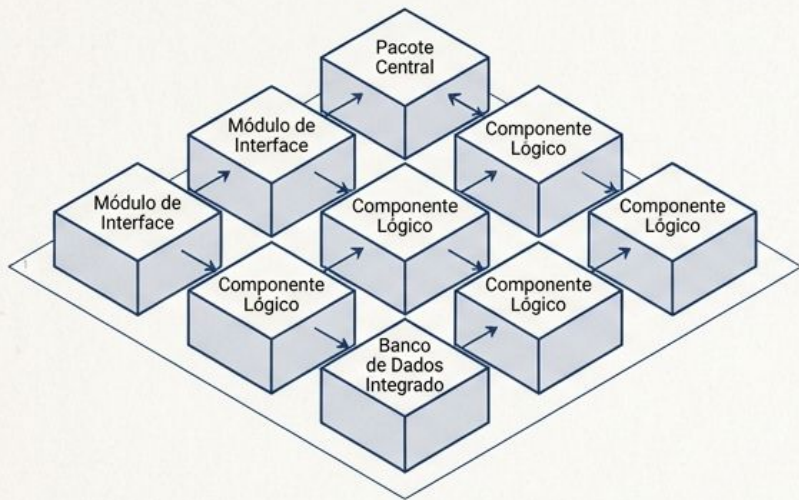
Relação em que uma parte precisa de outra.



— Linhas sólidas = interação entre componentes neste exemplo.

Estrutura Ideal pode não ser fácil de atingir

O Estado Ideal: Ordem Modular



1

Organização de Alto Nível



A divisão do sistema em pacotes, componentes e módulos.

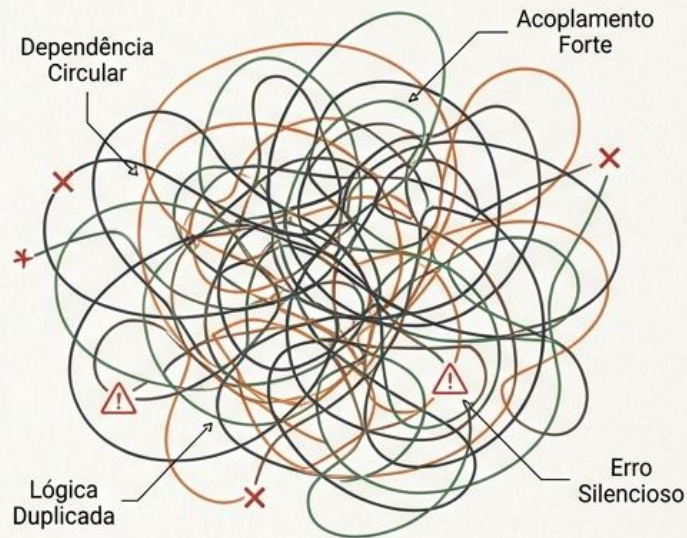
2

Decisões Difíceis de Reverter



Escolha de linguagens, bancos de dados e padrões de comunicação.

O Anti-Padrão: A Grande Bola de Lama



O Debate Tanenbaum-Torvalds: Até mesmo sistemas de extremo sucesso como o kernel monolítico do Linux acabam enfrentando o inchaço arquitetural previsto por críticos iniciais.

Arquitetura em Camadas

Organização Ilustrativa (Exemplo de Reserva)

1. Apresentação

Telas e formulários. Ex: Interface de Solicitação de Reserva



2. Aplicação

Coordenação dos casos de uso. Ex: Fluxo de Criar Reserva



Fluxo de Chamadas

3. Domínio

Regras de reservas e verificação de conflito de horários



4. Infraestrutura / Persistência

Acesso a serviços externos e banco de dados

Benefícios

- **Separação de Responsabilidades:** Organização clara e isolamento de conceitos.
- **Facilidade de Entendimento:** Cada camada possui um propósito bem definido.
- **Substituição com Contratos Estáveis:** Permite alterar implementações sem impactar outras camadas.

Desafios

- **Atravessamentos de Camadas:** Repasse de dados simples por múltiplas camadas sem lógica real.
- **Dependências Indevidas:** Risco de acoplamento direto ignorando abstrações.
- **Possível Sobrecarga (Overhead):** Complexidade adicional e menor desempenho em casos muito simples.

Camadas Lógicas vs. Tiers de Implantação

Layer / Camada Lógica (Organização do Código)

1. Apresentação

Interface do usuário, telas e formulários.

2. Aplicação

Coordenação de fluxos e casos de uso.

3. Domínio

Regras de negócio e lógicas centrais.

4. Infraestrutura / Persistência

Acesso a banco de dados e serviços externos.

Conceito Fundamental: Monólito Modular

Todas as 4 camadas lógicas acima podem ser compiladas e executadas juntas em uma **única unidade de implantação** (monólito).

Tier / Camada Física (Distribuição em Execução)

Tier 1: Cliente / Navegador

Execução no dispositivo do usuário final (Browser/Mobile).

↓ Rede (HTTP / HTTPS)

Tier 2: Servidor de Aplicação (Backend)

Ambiente de servidor onde o código do sistema executa.

↓ Rede / Conexão DB

Tier 3: Servidor de Dados

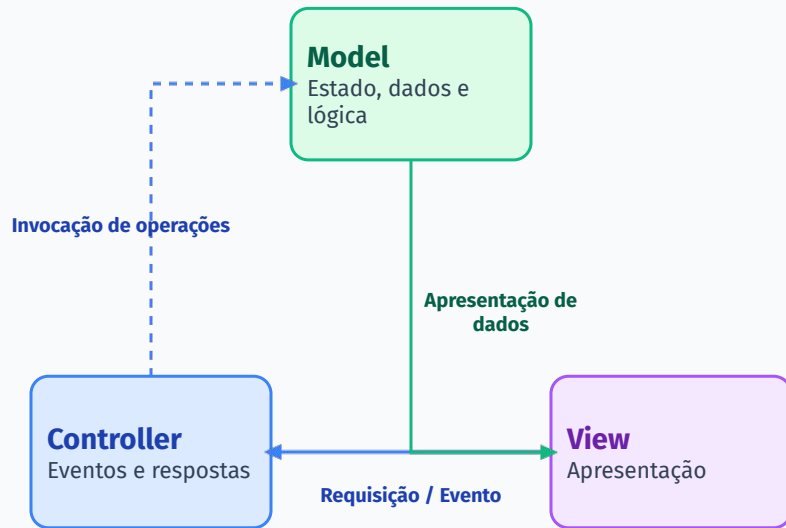
SGBD e persistência física isolada.

Pergunta

É possível ter 4 camadas lógicas em 1 única aplicação implantável?
Sim! Camadas organizam o código; Tiers definem onde ele roda.

Arquitetura MVC: Modelo, Visão e Controlador

Estrutura do Padrão MVC



Importante: MVC separa responsabilidades de interação e apresentação; não determina sozinho a implantação do sistema.

Exemplo Prático: "Consultar Sala Disponível"

C Controller: ReservaController

1. Recebe a requisição GET /salas/disponiveis?data=...
2. Invocação: chama o serviço/modelo de domínio para buscar salas.

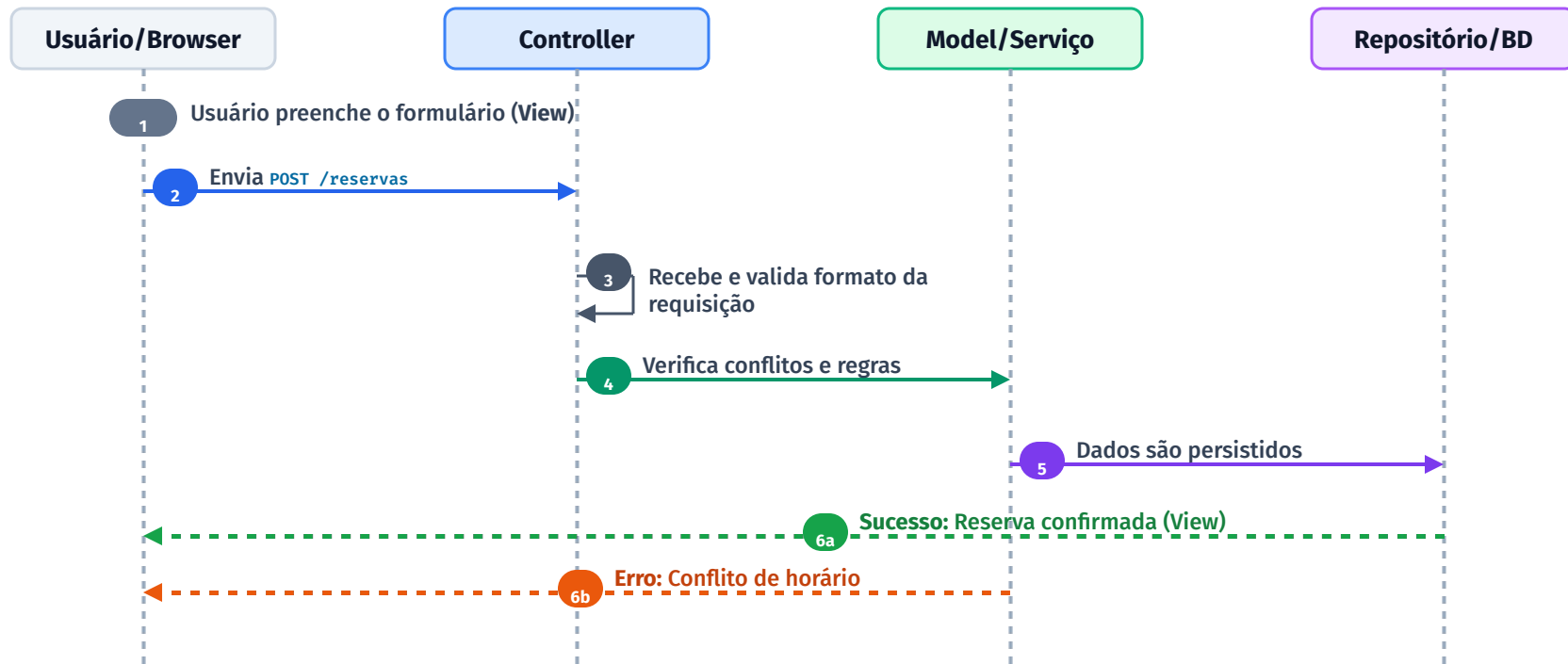
M Model: Sala & Agendamento

1. Valida regras de horário e consulta a disponibilidade das salas.
2. Retorna a lista de objetos Sala disponíveis.

V View: ConsultaSalasView / JSON Response

1. Formata os dados da sala recebidos para a interface visual.
2. Renderiza o HTML final ou envia a resposta JSON para o navegador.

MVC em Ação: Solicitar uma Reserva



Importante

Separação de Validações: O Controller valida o formato da requisição (ex: campos obrigatórios, tipos), enquanto o Modelo/Serviço valida as regras de negócio e políticas (ex: sobreposição de horários).

Exercício rápido: quem é responsável?

Classifique cada responsabilidade como **Model**, **View** ou **Controller**.

A Exibir calendário de disponibilidade.

B Receber requisição de cancelamento.

C Verificar se duas reservas conflitam.

D Mostrar mensagem “sala indisponível”.

E Aplicar regra de prazo máximo para cancelamento.

Exercício rápido: quem é responsável?

Classifique cada responsabilidade como **Model**, **View** ou **Controller**.

A Exibir calendário de disponibilidade.

View

B Receber requisição de cancelamento.

Controller

C Verificar se duas reservas conflitam.

Model

D Mostrar mensagem “sala indisponível”.

View

E Aplicar regra de prazo máximo para cancelamento.

Model

Variações de MVC

MVC — Model View Controller

Model: Dados e lógica **View:** Apresentação e interações **Controller:** Interpreta e coordena

Obs: View pode consultar Model diretamente.

MVP — Model View Presenter

Model: Dados e lógica **View:** Interface (sem acesso ao Model) **Presenter:** Intermediário total

Vantagem: Facilita testes isolados de apresentação.

MVVM — Model View View-Model

Model: Dados e regras **View:** Interface do usuário **ViewModel:** Estado para a View

Data Binding: Sincronização automática UI/ViewModel.

MVVM-C — Model View View-Model Coordinator

Estrutura: Baseada em MVVM **Coordinator:** Gerencia o fluxo de navegação

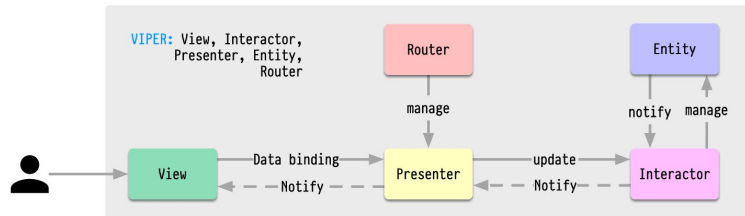
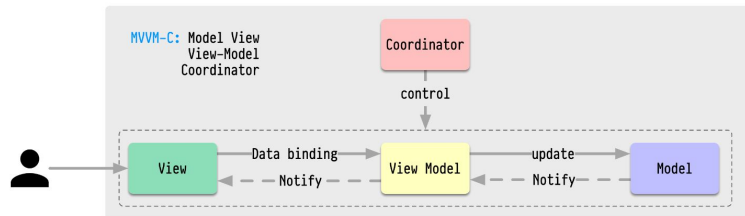
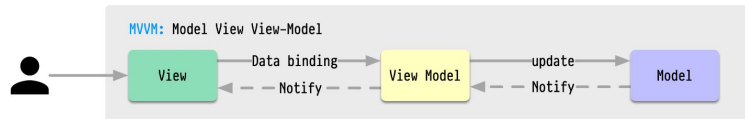
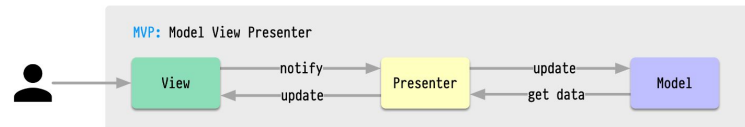
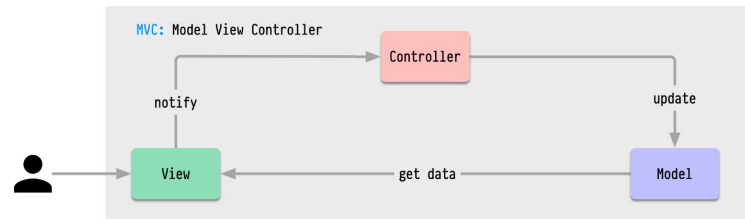
Vantagem: Separa navegação e suporta fluxos complexos.

VIPER — View Interactor Presenter Entity Router

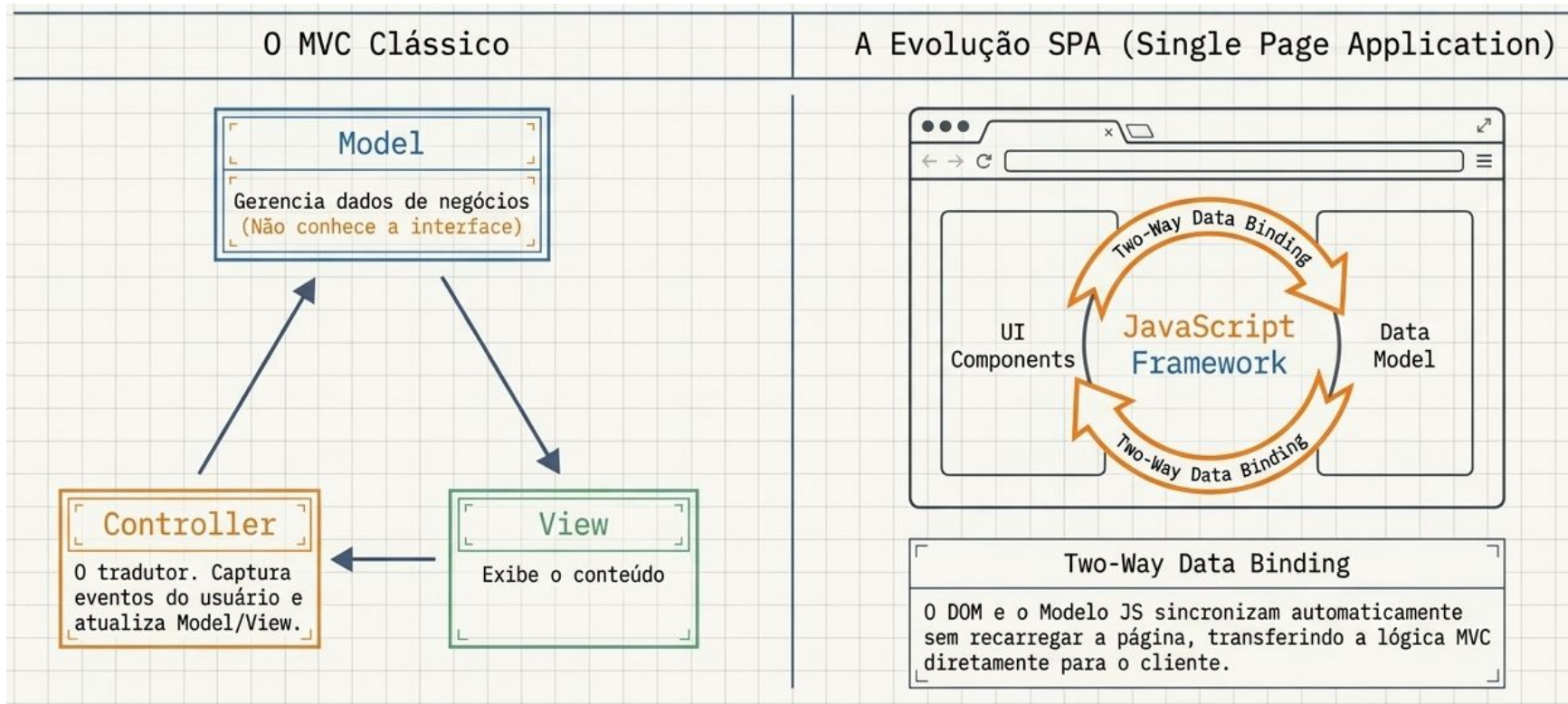
View: Interface **Interactor:** Casos de uso **Presenter:** Prepara apresentação

Entity: Conceitos de domínio **Router:** Navegação entre telas

Vantagem: Alta modularidade (maior complexidade).



MVC x SPA



Arquitetura Hexagonal

O Problema

Regras de negócio acopladas diretamente a **frameworks UI**, queries **SQL** e provedores externos, dificultando testes e evolução.

Princípio Fundamental

A infraestrutura pode mudar sem impor alterações às regras de negócio.
Interior/exterior referem-se a responsabilidades e dependências, não a máquinas físicas.

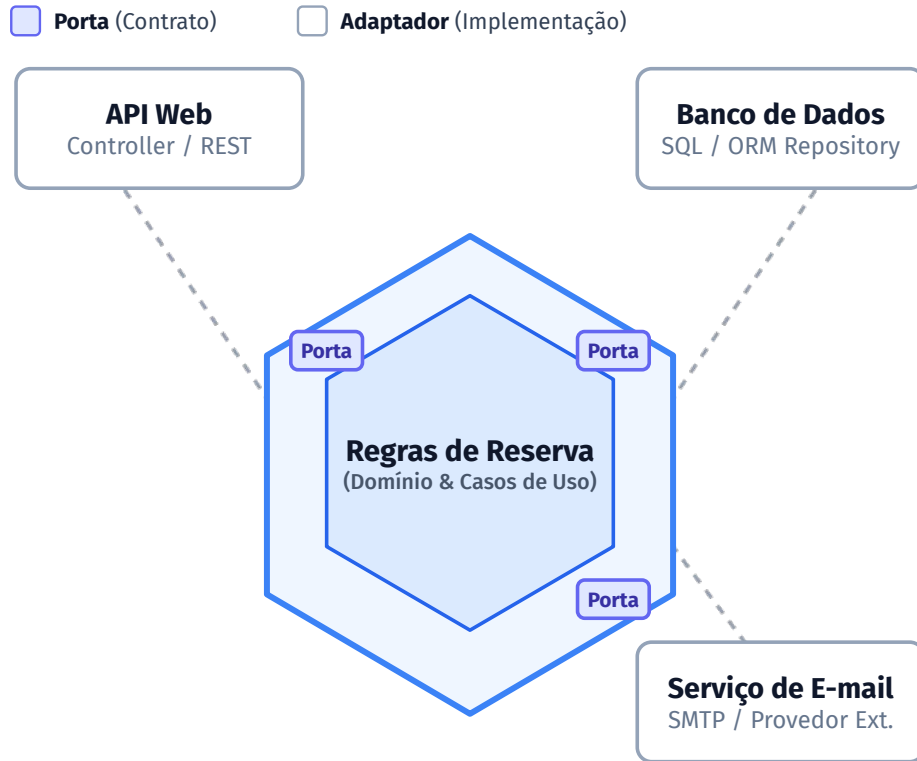
Conceitos Fundamentais

Núcleo: Domínio e Casos de Uso.

Portas: Contratos/Interfaces de interação.

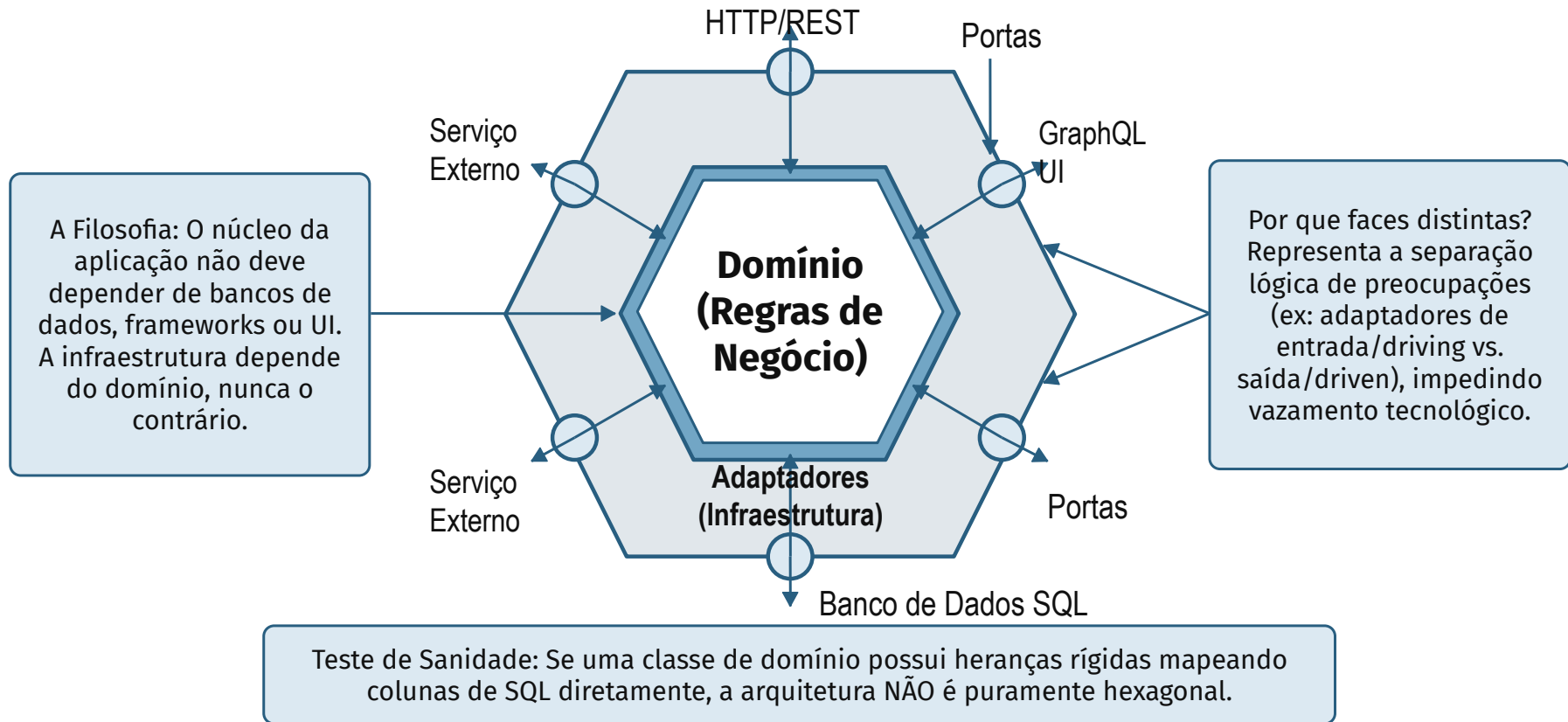
Adaptadores: Implementações concretas.

Nota: O hexágono não limita a 6 componentes.



Pergunta: Como testar a regra de negócio sem subir o banco de dados ou a API?

Arquitetura Hexagonal: Visão Geral



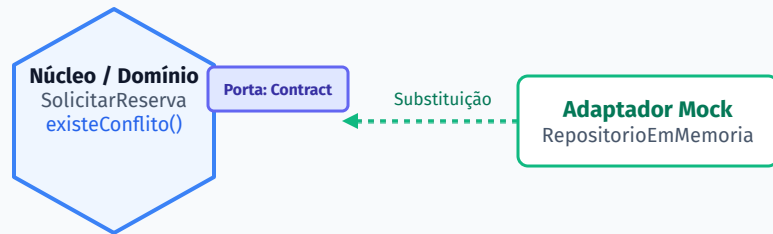
Inversão de Dependências

A **Inversão de Dependências** permite testar casos de uso sem infraestrutura externa (banco/API) e facilita a troca de persistência.

Ambiente de Produção



Ambiente de Testes



Isolamento de Regras de Negócio & Inversão de Dependência

```
interface RepositorioReservas { buscarPorIntervalo(sala, inicio, fim): Lista<Reserva> }

class SolicitarReserva {
  function executar(sala, inicio, fim) {
    if (existeConflito(sala, inicio, fim)) throw Error("Conflito de horário!"); // Regra no Núcleo
    this.repo.salvar(novaReserva(sala, inicio, fim));
  }
}
```

Command Query Responsibility Segregation (CQRS)

Separar escrita e leitura quando necessário

Conceitos Fundamentais

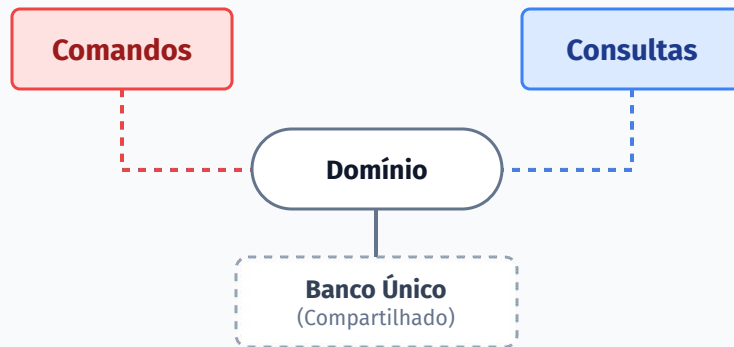
Comando (Write): Altera o estado do sistema.

`CriarReserva, CancelarReserva`

Consulta (Read): Retorna dados sem alterar o estado.

`ConsultarDisponibilidade`

Visão Arquitetural (Separação Lógica)



Atenção & Trade-offs

Complexidade: Não exige dois bancos, múltiplos serviços ou mensageria/eventos em toda implementação.

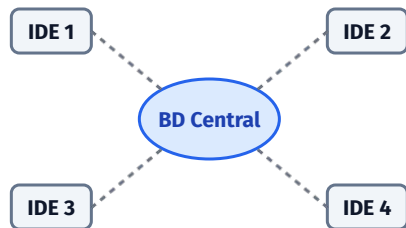
Pergunta Prática: Uma aplicação CRUD pequena precisa obrigatoriamente de CQRS? **Não.** Evite complexidade desnecessária.

Importante: Separe modelos de escrita e leitura apenas quando suas necessidades divergirem.

Três outros padrões: Repositório, Duto-Filtro e Serviços

Visão geral de arquiteturas alternativas e suas topologias características.

1. Repositório



Conceito: Ferramentas independentes compartilham um armazenamento de dados central comum.

Topologia: Hub central (estrela/concentrador).

Desafio/Risco: Alta concentração de dependências no esquema de dados central e gargalo de acesso.

2. Duto-Filtro (Pipe-Filter)

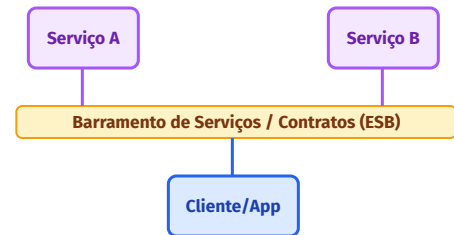


Conceito: Processamento de dados estruturado em uma série de etapas autônomas e sequenciais.

Topologia: Pipeline linear unidirecional.

Aplicação: Adequado para transformações de dados em fluxo, compiladores e processamento em lote.

3. Orientado a Serviços (SOA)



Conceito: Sistemas corporativos expõem funcionalidades fracamente acopladas por contratos.

Topologia: Malha distribuída de serviços conectados por interfaces bem definidas.

Desafio/Risco: Alta complexidade de integração, governança e orquestração de rede.

Monólito não significa software desorganizado

Monólito Modular (Aplicação Unificada)

Processo Único Implantável (Deployable Unit)

Módulo Reservas

- Regras Nég.
- Interfaces

Módulo Salas

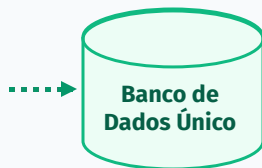
- Regras Nég.
- Interfaces

Módulo Usuários

- Regras Nég.
- Interfaces

Módulo Notificações

- Regras Nég.
- Interfaces



Potenciais Vantagens

- **Simplicidade de Implantação:** Apenas uma unidade para publicar, monitorar e gerenciar pipeline CI/CD.
- **Chamadas Locais:** Comunicação em memória entre módulos, garantindo alta performance e zero latência de rede.
- **Operações e Transações Directas:** Uso de transações ACID nativas do banco de dados sem necessidade de sagas complexas.

Limites Possíveis

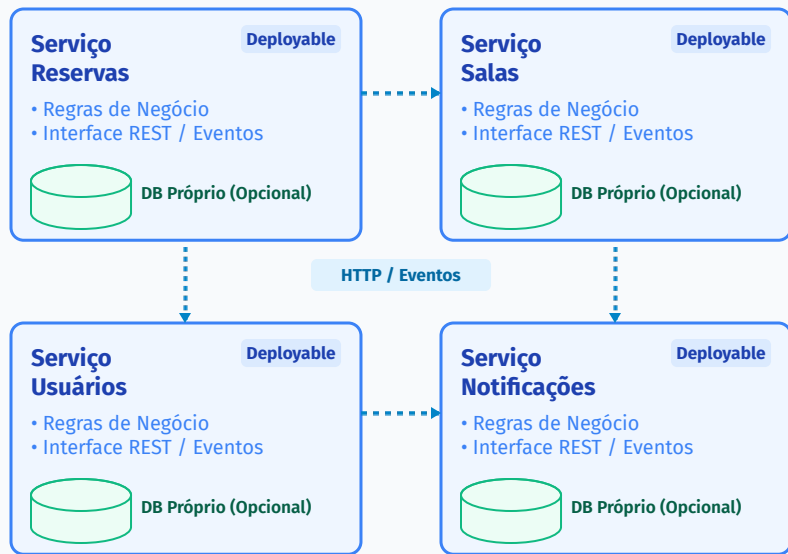
- **Acoplamento Mal Gerido:** Risco de o monólito se transformar em uma "massa sem estrutura" ("Big Ball of Mud").
- **Releases Compartilhadas:** Qualquer alteração exige reimplantação de toda a aplicação unificada.
- **Escalabilidade Conjunta:** Necessidade de escalar toda a aplicação mesmo que apenas um módulo sofra alta carga.

Módulos não precisam virar processos independentes para ter separação de responsabilidades.

Microsserviços: Autonomia com Custo Distribuído

Serviços autônomos por capacidades de negócio, comunicando-se por rede com custos operacionais próprios.

Arquitetura de Serviços Independentes



Potenciais Ganhos

- **Implantação Independente:** Prototipação, deploy e atualização de serviços sem impactar o sistema global.
- **Escalabilidade Seletiva:** Alocação de recursos especificamente nos módulos com maior demanda.
- **Autonomia de Equipes:** Times independentes trabalhando com tecnologias e ciclos de vida próprios.

Novos Custos e Desafios

- **Complexidade de Rede & Operação:** Latência, falhas parciais e alta demanda de monitoramento.
- **Consistência Distribuída:** Perda de transações ACID nativas, exigindo padrões como Sagas.
- **Gestão de Fronteiras:** Garantir que os limites dos serviços correspondam a responsabilidades coesas.

Microsserviços não são apenas rotas ou classes. Benefícios não são automáticos e exigem maturidade.

Microserviços: Componentes

1. Balanceador de Carga

Distribui o tráfego de entrada entre os múltiplos serviços de backend, evitando sobrecarga.

2. CDN (Conteúdo Estático)

Servidores distribuídos que armazenam conteúdos estáticos para entrega rápida antes de acessar o backend.

3. Gateway de API

Recebe requisições e as encaminha aos serviços adequados, comunicando com autenticação e descoberta.

4. Provedor de Identidade

Responsável pelo controle unificado de autenticação e autorização dos usuários na plataforma.

5. Registro e Descoberta

Registra e localiza microserviços ativos dinamicamente para permitir a comunicação via API Gateway.

6. Gerenciamento

Componente responsável pelo monitoramento contínuo da saúde e observabilidade dos serviços.

7. Microserviços

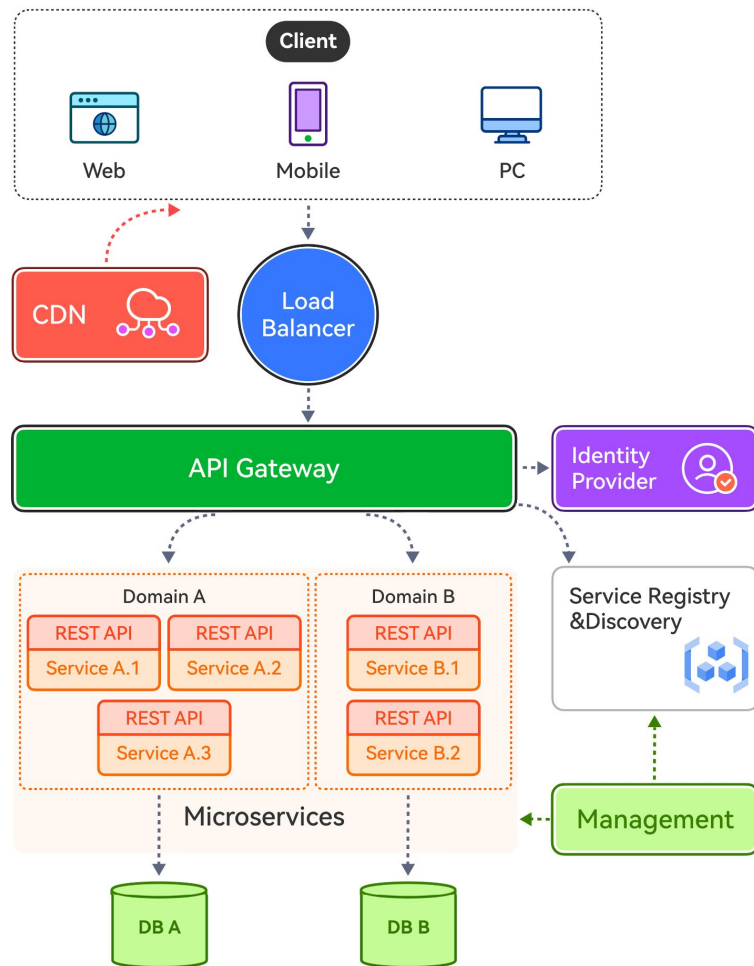
Projetados por domínios de negócio independentes, onde cada domínio possui seu próprio banco de dados.

Visão de Integração

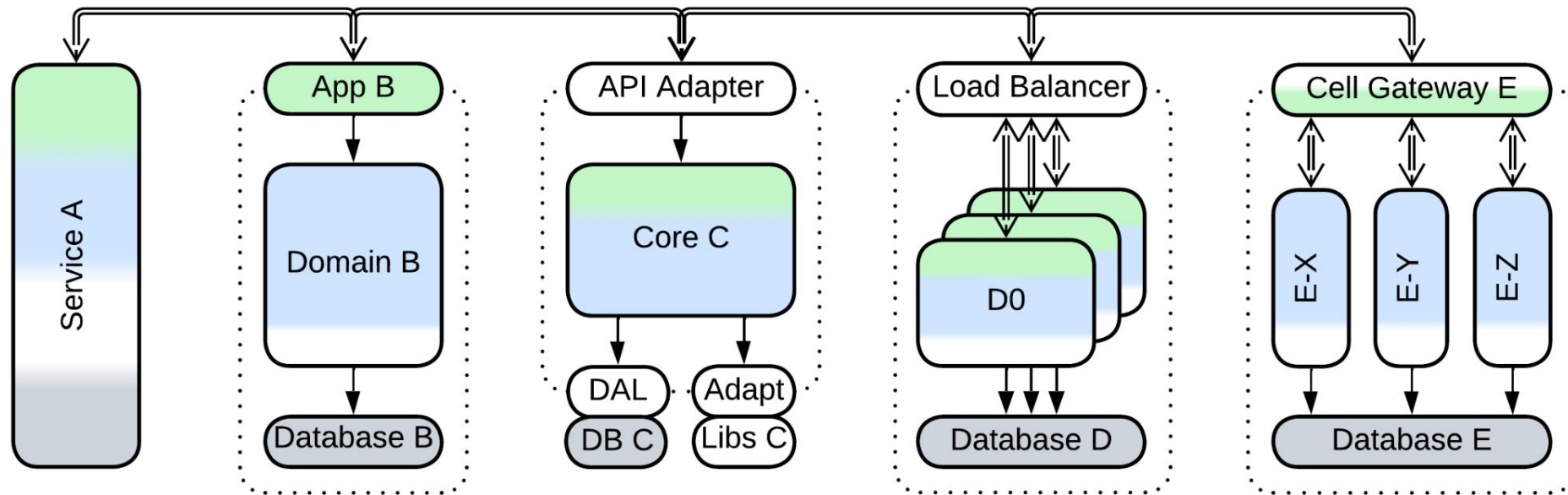
Cada componente desempenha um papel fundamental para garantir alta disponibilidade, segurança e escalabilidade.

Typical Microservice Architecture

ByteByteGo



Padrões Arquiteturais de Serviços



**Monolithic
Service**

**Layered
Service**

**Hexagonal
Service**

**Scaled
Service**

Cell

Variantes da estrutura interna de serviços. Fonte: [Metapatterns](#).

Boas práticas

1. Projetar para Falhas

Antecipe falhas usando Circuit Breaker, Bulkhead e Degradação Controlada.

2. Serviços Pequenos

Garanta responsabilidades únicas e bem definidas para cada serviço.

3. Protocolos Leves

Adote padrões eficientes de comunicação como REST, gRPC e Message Brokers.

4. Descoberta de Serviços

Localize componentes na rede com Consul, Eureka ou Kubernetes.

5. Propriedade dos Dados

Monitore e gerencie dados de forma isolada por microserviço.

6. Padrões de Resiliência

Melhore disponibilidade com políticas de Retry, Caching e Rate Limiting.

7. Segurança em Níveis

Proteja a comunicação e implemente controles em todas as camadas.

8. Logs Centralizados

Consolide registros para facilitar diagnósticos e monitoramento.

9. Containerização

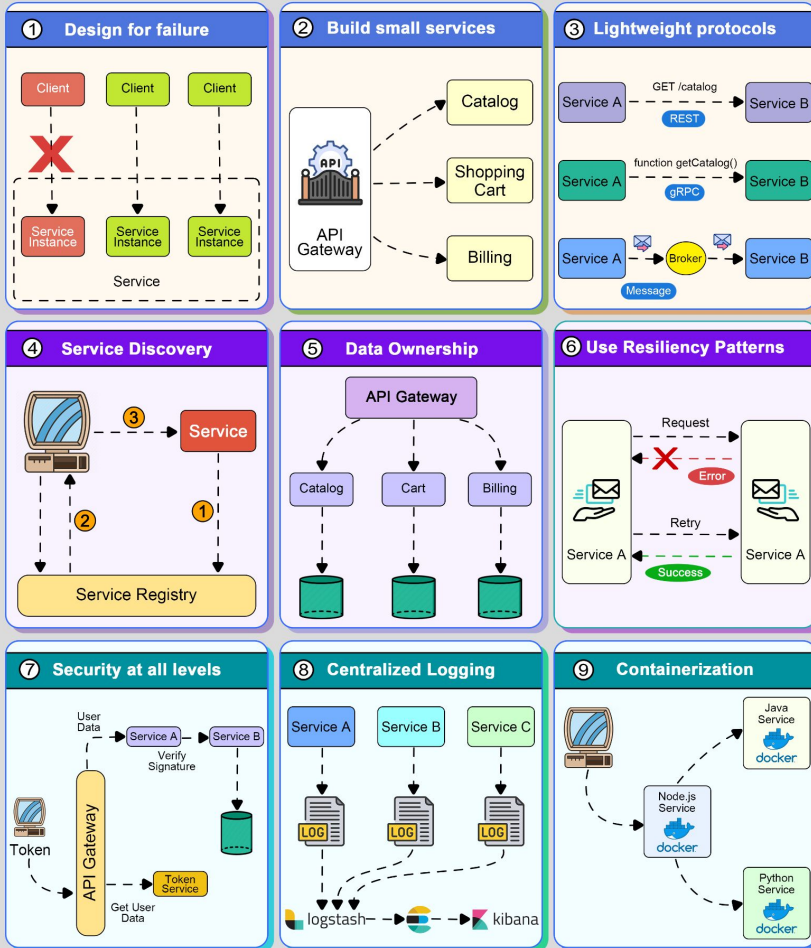
Isole e escale ambientes com Docker e Kubernetes.

Objetivo Arquitetural
Autonomia, Resiliência & Escalabilidade

Importante

O sucesso dos microserviços exige autonomia, resiliência, segurança e monitoramento robusto, indo além da simples divisão de código.

9 Best Practices For Building Microservices



Estudo de Caso

O Sistema (Visão Geral & Funcionalidades)



Contexto Operacional

Universidade, aplicação Web, equipe de 4 profissionais e perspectiva de crescimento gradual.

Requisitos Funcionais (RF):

RF01 Consultar disponibilidade: busca rápida por salas/horários vagos.

RF02 Solicitar reserva: submissão de novos agendamentos.

RF03 Impedir reservas conflitantes: validação consistente de choque de horários.

RF04 Cancelar reserva: liberação de horários previamente alocados.

RF05 Notificar confirmação/cancelamento: envio de alertas aos usuários.

Qualidades e Restrições (RNF)

RNF01 Desempenho & Tempo de Resposta

Consulta em até 2 segundos sob carga normal definida.

RNF02 Testabilidade Decoplada

Regras de negócio testáveis sem necessidade da interface web.

RNF03 Manutenibilidade & Evolução

Evolução do sistema com impacto limitado entre módulos.

RNF04 Simplicidade Operacional

Baixa complexidade operacional e de implantação na fase inicial.

Conclusão

Uma boa arquitetura **atende aos requisitos** com complexidade e riscos proporcionais ao problema.

01. Estrutura & Responsabilidades

Arquitetura estrutura responsabilidades e dependências.

02. Decisões Orientadas

Requisitos e qualidade orientam decisões.

03. Flexibilidade de Padrões

Padrões oferecem alternativas, não receitas.

04. Trade-offs do Desacoplamento

Desacoplamento traz flexibilidade e também custo.

Princípio Fundamental: A decisão deve ser justificável no contexto.

Referências

Bibliografia Básica da Disciplina

Roger S. Pressman & Bruce R. Maxim

Engenharia de Software: Uma Abordagem Profissional

Capítulos 9 (completo)

Ian Sommerville

Engenharia de Software
Capítulo 7 (completo)

Referências Complementares de Apoio

Marco Tulio Valente

Engenharia de Software Moderna
Capítulo 7 (completo)

<https://engsoftmoderna.info/cap5.html>

ByteByteGo: 10 System Design Tradeoffs You Cannot Ignore

<https://bytebytego.com/guides/10-system-design-tradeoffs-you-cannot-ignore/>

Dúvidas e Discussão