IT VALLEY CLEAN ARCHITECTURE

DA TEORIA À PRÁTICA

CARLOS VIANA

Josué 1:9

“ Não fui eu que lhe ordenei?
Seja forte e corajoso!
Não se apavore nem desanime,
pois o Senhor, o seu Deus,
estará com você por onde você andar. ”

**app/domain/cl# Arquitetura Limpa IT Valley

Um Guia Completo para Estudantes

Introdução: Bem-vindo à Arquitetura IT Valley!

Meu querido estudante, seja bem-vindo ao mundo da Arquitetura Limpa IT Valley! Este documento vai te ensinar uma das formas mais elegantes e organizadas de estruturar sistemas de software.

O que você vai aprender aqui:

Ao final desta jornada, você será capaz de construir sistemas robustos, testáveis e fáceis de manter. Você dominará cada camada da nossa arquitetura e saberá exatamente onde colocar cada linha de código.

🎯 ANOTE AQUI O QUE VOCÊ VAI DOMINAR:

1. API (Interface Externa):

- Responsabilidade: _____
- O que pode fazer: _____
- O que NÃO pode fazer: _____

2. Service (Orquestrador):

- Responsabilidade: _____
- O que pode fazer: _____
- O que NÃO pode fazer: _____

3. DTO/Schemas (Contratos):

- Responsabilidade: _____
- O que pode fazer: _____
- O que NÃO pode fazer: _____

4. Domain (Coração do Sistema):

- Responsabilidade: _____
- O que pode fazer: _____
- O que NÃO pode fazer: _____

5. Factory (Fábrica):

- Responsabilidade: _____
- O que pode fazer: _____
- O que NÃO pode fazer: _____

6. Repository (Persistência):

- Responsabilidade: _____
- O que pode fazer: _____
- O que NÃO pode fazer: _____

📝 **Suas Anotações Importantes:**

_____

_____

_____

_____

_____

Dicas de Estudo:

- Leia cada seção com calma, meu querido
- Anote dúvidas e volte depois
- Pratique os exemplos no seu computador
- Não tenha pressa - arquitetura se aprende devagar

Agora vamos começar nossa jornada! 🚀

Sumário

---

1. Por que Arquitetura Importa?

Meu querido, imagine que você está construindo uma casa. Você pode simplesmente empilhar tijolos sem pensar, ou pode fazer um projeto arquitetônico primeiro. Qual das duas vai resultar numa casa mais sólida e bonita?

No desenvolvimento de software, é a mesma coisa. Sem uma arquitetura bem definida, seu código vira uma bagunça que ninguém consegue entender ou modificar.

Problemas de código sem arquitetura:

- Quando você muda uma coisa, quebra três outras
- É impossível testar as funcionalidades isoladamente
- Novos desenvolvedores demoram semanas para entender o código
- Bugs aparecem em lugares inesperados
- Adicionar novas funcionalidades vira um pesadelo

Com uma boa arquitetura:

- Mudanças ficam isoladas e controladas
- Cada parte pode ser testada independentemente
- O código fica fácil de entender e modificar
- Bugs ficam contidos em suas respectivas camadas
- Novas funcionalidades são simples de adicionar

---

2. O Conceito de Camadas

Pense numa empresa bem organizada, meu querido. Você tem diferentes departamentos:

- **Atendimento ao Cliente**: Fala com o público
- **Vendas**: Processa pedidos
- **Estoque**: Gerencia produtos

- **Contabilidade**: Cuida do dinheiro

Cada departamento tem sua responsabilidade específica e se comunica com os outros de forma organizada.

No software, fazemos igual:

```
┌─────────────────────────┐
│   Camada API    │ ← Fala com o mundo exterior (como Atendimento)
├─────────────────────────┤
│ Camada Service  │ ← Processa regras de negócio (como Vendas)
├─────────────────────────┤
│ Camada Domain   │ ← Guarda as regras importantes (como Políticas da Empresa)
├─────────────────────────┤
│ Camada Repository│ ← Salva e busca dados (como Arquivo)
└─────────────────────────┘
```

Benefícios das Camadas:

- **Organização**: Cada coisa no seu lugar
- **Responsabilidade clara**: Cada camada tem um papel específico
- **Facilita manutenção**: Mudou algo na API? Só mexe na camada API
- **Testabilidade**: Testa cada camada separadamente

---

3. O Problema do Acoplamento

Acoplamento é quando as coisas estão "grudadas" umas nas outras. Como aqueles fones de ouvido que sempre enroscam no bolso!

Exemplo de código acoplado (RUIM):

```
python
```

```python
# API fazendo tudo misturado - PÉSSIMO!
@app.post("/clientes")
def criar_cliente(request):
    # API acessando banco direto (acoplado!)
    if db.query("SELECT * FROM clientes WHERE email = ?", request.email):
        return {"erro": "Email já existe"}

    # API fazendo validação (responsabilidade errada!)
    if len(request.nome) < 2:
        return {"erro": "Nome muito curto"}

    # API montando SQL (muito acoplado!)
    db.execute("""
        INSERT INTO clientes (nome, email, status)
        VALUES (?, ?, 'ativo')
    """, request.nome, request.email)

    return {"sucesso": True}
```

Problemas desse código:

- API conhece detalhes do banco de dados
- Se mudar a tabela, quebra a API
- Impossível testar sem banco real
- Regras de negócio espalhadas por todo lugar
- Se mudar validação, tem que mexer na API

---

4. A Solução: Desacoplamento

Desacoplamento é separar as responsabilidades e fazer cada parte conversar através de "contratos" bem definidos.

Exemplo desacoplado (BOM):

```python
```

```
# API só coordena
@app.post("/clientes")
def criar_cliente(request):
    try:
        cliente = cliente_service.criar(request, repo)
        return mapper.to_response(cliente)
    except ValueError as e:
        return {"erro": str(e)}

# Service cuida das regras de negócio
def criar(request, repo):
    if repo.existe_email(request.email):
        raise ValueError("Email já existe")

    cliente = factory.criar_cliente(request)
    repo.salvar(cliente)
    return cliente

# Repository cuida do banco
def salvar(cliente):
    self.db.execute("INSERT INTO...", cliente.nome, cliente.email)
```

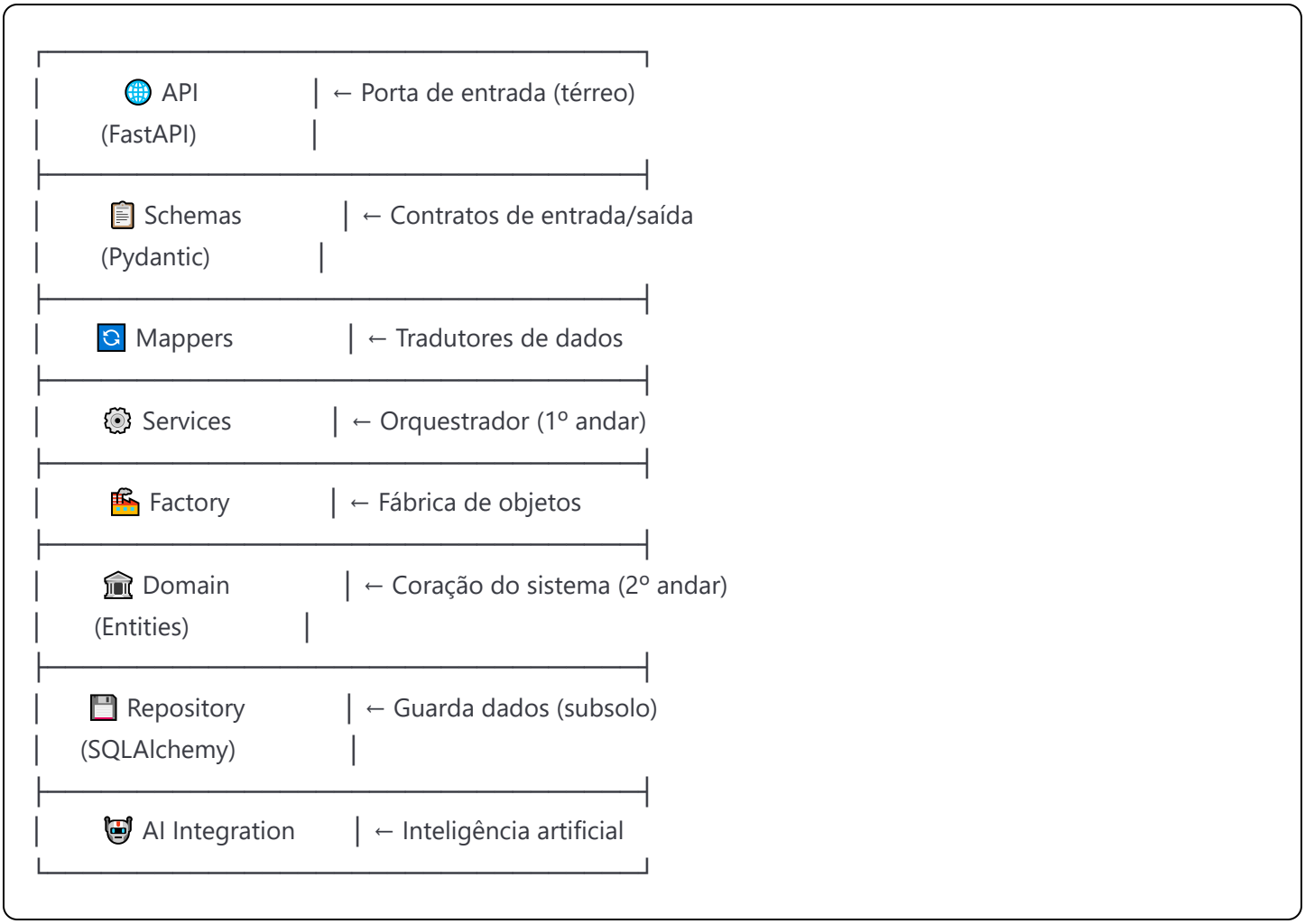
Vantagens:

- API não conhece banco de dados
- Service não conhece detalhes de persistência
- Cada parte pode ser testada isoladamente
- Mudanças ficam contidas

---

5. Nossa Arquitetura IT Valley

Agora vou te mostrar nossa arquitetura completa, meu querido. É como um prédio bem organizado:



Fluxo de dados (de cima para baixo):

1. **API** recebe requisição HTTP
2. **Schemas** validam dados de entrada
3. **Service** orquestra o processo
4. **Factory** cria objetos do domínio
5. **Domain** aplica regras de negócio
6. **Repository** salva no banco

Fluxo de resposta (de baixo para cima):

1. **Repository** retorna dados
2. **Domain** processa regras
3. **Service** coordena retorno
4. **Mapper** converte para resposta
5. **API** retorna HTTP

---

6. As Camadas em Detalhes

Vou explicar cada camada como se você fosse trabalhar nela amanhã, meu querido.

🌐 Camada API (Interface Externa)

Responsabilidade: Ser a "porta de entrada" do sistema. Como um recepcionista educado.

O que faz:

- Recebe requisições HTTP (GET, POST, PUT, DELETE)
- Valida se os dados estão no formato correto
- Chama o Service apropriado
- Retorna resposta HTTP formatada
- Trata erros de forma amigável

Analogia: É o atendente da loja que recebe o cliente, entende o que ele quer, chama o vendedor certo e entrega o produto.

```python
@router.post("/clientes", response_model=ClienteResponse)
def criar_cliente(dto: ClienteCreateRequest, repo=Depends(repo_dep)):
    # 1. Recebeu dados (dto)
    # 2. Chama service
    cliente = service.criar_cliente(dto, repo)
    # 3. Converte resposta
    return mapper.to_public(cliente)
```

📋 Camada Schemas (Contratos)

Responsabilidade: Definir "contratos" de entrada e saída. Como um formulário bem estruturado.

O que faz:

- Define formato dos dados de entrada (Request)
- Define formato dos dados de saída (Response)
- Valida tipos, tamanhos, formatos
- Documenta automaticamente a API

Analogia: São os formulários padronizados que o cliente preenche e os recibos formatados que ele recebe.

```python
class ClienteCreateRequest(BaseModel):
    nome: str = Field(min_length=2, max_length=100)
    email: EmailStr
    telefone: Optional[str] = None

class ClienteResponse(BaseModel):
    id: str
    nome: str
    email: str
    status: str
```

🔄 Camada Mappers (Tradutores)

Responsabilidade: Traduzir entre diferentes formatos de dados.

O que faz:

- Converte Entity (domínio) para Response (API)
- Permite diferentes "visões" dos mesmos dados
- Protege dados internos de vazarem para fora

Analogia: É o tradutor que converte a linguagem técnica interna para a linguagem que o cliente entende.

```python
```

```python
class ClienteMapper:
    @staticmethod
    def to_public(cliente: Cliente) -> ClienteResponse:
        return ClienteResponse(
            id=cliente.id,
            nome=cliente.nome,
            email=cliente.email,
            status=cliente.status
        )

    @staticmethod
    def to_display(cliente: Cliente) -> ClienteDisplayResponse:
        return ClienteDisplayResponse(
            nome=cliente.nome,
            cargo=cliente.cargo  # Só nome e cargo para tela simples
        )
```

⚙️ Camada Service (Orquestrador)

Responsabilidade: Coordenar casos de uso complexos. É o "gerente" que organiza tudo.

O que faz:

- Orquestra fluxos de negócio
- Coordena entre diferentes partes
- Valida regras que dependem de dados externos
- Controla transações

Analogia: É o gerente que coordena: "Primeiro consulta estoque, depois calcula desconto, se tudo ok, registra venda".

```python
```

```python
def criar_cliente(dto: ClienteCreateRequest, repo) -> Cliente:
    # 1. Validações que dependem do repositório
    email = email_from(dto)
    if repo.existe_email(email):
        raise ValueError("Email já existe")

    # 2. Cria entidade via Factory
    cliente = criar_cliente(dto)

    # 3. Persiste
    repo.add(cliente)

    return cliente
```

🏭 Camada Factory (Fábrica)

Responsabilidade: Criar objetos do domínio de forma consistente.

O que faz:

- Única porta de entrada para criar entidades
- Centraliza validações de criação
- Converte dados externos para objetos internos
- Fornece helpers para Service extrair dados

Analogia: É a fábrica que monta carros. Você dá as especificações e ela entrega o carro pronto e testado.

```python
```

```python
class ClienteFactory:
    @staticmethod
    def make_client(dto: Any) -> Cliente:
        nome = _get(dto, "nome")
        email = _get(dto, "email")

        # Validações
        if not nome or len(nome.strip()) < 2:
            raise ValueError("Nome muito curto")

        # Cria entidade completa
        return Cliente(
            id=uuid4().hex,
            nome=nome.strip(),
            email=email.strip(),
            status="ativo",
            created_at=datetime.utcnow()
        )

    @staticmethod
    def email_from(dto: Any) -> str:
        return _get(dto, "email")  # Helper para Service
```

🏛️ Camada Domain (Coração)

Responsabilidade: Guardar as regras de negócio mais importantes. É a "alma" do sistema.

O que faz:

- Define entidades (Cliente, Produto, Pedido)
- Implementa comportamentos importantes
- Garante integridade dos dados
- Aplica regras de negócio puras

Analogia: São as regras fundamentais da empresa que nunca mudam, como "cliente deve ter nome" ou "preço não pode ser negativo".

```python
```

```python
@dataclass
class Cliente:
    id: str
    nome: str
    email: str
    status: str = "ativo"

    def ativar(self) -> None:
        if self.status == "banido":
            raise ValueError("Cliente banido não pode ser ativado")
        self.status = "ativo"

    def desativar(self) -> None:
        self.status = "inativo"

    def aplicar_atualizacao_from_any(self, data: Any) -> None:
        # Aplica mudanças de qualquer fonte de dados
        novo_nome = _get(data, "nome")
        if novo_nome and len(novo_nome.strip()) >= 2:
            self.nome = novo_nome.strip()
```

💾 Camada Repository (Armazenamento)

Responsabilidade: Guardar e buscar dados. É o "arquivo" do sistema.

O que faz:

- Salva entidades no banco de dados
- Busca entidades por diferentes critérios
- Converte entre Entity (domínio) e Model (banco)
- Isola detalhes do banco de dados

Analogia: É o arquivista que sabe exatamente onde guardar cada documento e como encontrá-lo depois.

```python
```

```python
class ClienteRepository:
    def add(self, cliente: Cliente) -> None:
        model = self._to_model(cliente)
        self.session.add(model)
        self.session.commit()

    def get_by_email(self, email: str) -> Optional[Cliente]:
        model = self.session.query(ClienteModel).filter_by(email=email).first()
        return self._to_entity(model) if model else None

    def _to_entity(self, model: ClienteModel) -> Cliente:
        # Converte banco → domínio
        return Cliente(
            id=model.id,
            nome=model.nome,
            email=model.email,
            status=model.status
        )

    def _to_model(self, entity: Cliente) -> ClienteModel:
        # Converte domínio → banco
        return ClienteModel(
            id=entity.id,
            nome=entity.nome,
            email=entity.email,
            status=entity.status
        )
```

🤖 Camada AI Integration (Inteligência)

Responsabilidade: Integrar com serviços de IA externos.

O que faz:

- Define contratos para chamadas de IA
- Implementa adapters para diferentes provedores
- Processa resultados de IA
- Mantém isolamento de IA do resto do sistema

Analogia: É o consultor especializado que você chama quando precisa de uma análise específica.

```python
```

```python
class ClienteAIClient:
    def analisar_perfil(self, payload: ClienteAIPayload) -> ClienteAIResponse:
        # Chama serviço de IA externo
        response = self.ai_service.analyze(payload)
        return ClienteAIResponse(
            score_engajamento=response.score,
            categoria=response.category
        )
```

---

7. O Que Pode e Não Pode em Cada Camada

Agora vou te dar as regras claras, meu querido. É como um código de trânsito - precisa seguir para funcionar.

🌐 API - PODE e NÃO PODE

✅ PODE:

- Receber requisições HTTP
- Validar formato básico (Pydantic faz isso)
- Chamar Services
- Usar Mappers para converter Entity → Response
- Tratar exceções e retornar erros HTTP amigáveis
- Configurar dependency injection simples

❌ NÃO PODE:

- Acessar banco de dados diretamente
- Implementar regras de negócio
- Conhecer detalhes de como Entity funciona
- Fazer cálculos complexos
- Acessar campos do DTO diretamente (usar helpers)
- Instanciar conexões de banco

```python
```

```python
# ✅ CORRETO
@router.post("/clientes")
def criar(dto: ClienteCreateRequest, repo=Depends(repo_dep)):
    cliente = service.criar_cliente(dto, repo)
    return mapper.to_public(cliente)

# ❌ ERRADO
@router.post("/clientes")
def criar(dto: ClienteCreateRequest):
    # ERRO: API acessando banco
    if db.query("SELECT * FROM clientes WHERE email = ?", dto.email):
        return {"erro": "Email existe"}

    # ERRO: API fazendo validação
    if len(dto.nome) < 2:
        return {"erro": "Nome curto"}
```

📋 Schemas - PODE e NÃO PODE

✅ PODE:

- Definir estruturas de Request e Response
- Validar tipos, tamanhos, formatos
- Usar Field() para documentação
- Ter validations automáticas do Pydantic
- Definir diferentes DTOs para contextos diferentes

❌ NÃO PODE:

- Implementar lógica de negócio
- Acessar banco de dados
- Fazer cálculos complexos
- Depender de outras camadas (exceto tipos básicos)

```python
```

```python
# ✅ CORRETO
class ClienteCreateRequest(BaseModel):
    nome: str = Field(min_length=2, max_length=100)
    email: EmailStr
    telefone: Optional[str] = None

# ❌ ERRADO
class ClienteCreateRequest(BaseModel):
    nome: str
    email: str

    def validar_email_unico(self):  # ERRO: regra de negócio
        # Não pode acessar banco aqui!
        pass
```

⚙️ Service - PODE e NÃO PODE

✅ PODE:

- Orquestrar casos de uso
- Chamar Factory para criar entidades
- Usar Repository para persistir/buscar
- Coordenar entre diferentes partes
- Validar regras que dependem de dados externos
- Usar helpers da Factory para extrair dados

❌ NÃO PODE:

- Acessar campos do DTO diretamente (dto.nome)
- Conhecer detalhes de banco de dados
- Instanciar entidades diretamente
- Implementar validações que a Entity pode fazer
- Retornar DTOs (retorna Entities)

```python
```

```python
# ✅ CORRETO
def criar_cliente(dto: ClienteCreateRequest, repo) -> Cliente:
    email = email_from(dto)  # Helper da Factory
    if repo.existe_email(email):
        raise ValueError("Email já existe")

    cliente = criar_cliente(dto)  # Factory cria
    repo.add(cliente)
    return cliente

# ❌ ERRADO
def criar_cliente(dto: ClienteCreateRequest, repo) -> Cliente:
    if repo.existe_email(dto.email):  # ERRO: acessou dto.email
        raise ValueError("Email já existe")

    # ERRO: criou entidade diretamente
    cliente = Cliente(nome=dto.nome, email=dto.email)
    repo.add(cliente)
    return cliente
```

🏭 Factory - PODE e NÃO PODE

✅ PODE:

- Ser única porta de criação de entidades
- Fazer validações de criação
- Extrair dados de DTOs usando _get()
- Fornecer helpers para Service (email_from, id_from)
- Converter tipos de dados
- Gerar IDs únicos

❌ NÃO PODE:

- Acessar banco de dados
- Depender de Repository
- Fazer validações que precisam de dados externos
- Retornar entidades parciais
- Ter múltiplas formas de criar a mesma entidade

python

```python
# ✅ CORRETO
def criar_cliente(dto: Any) -> Cliente:
    nome = _get(dto, "nome")
    email = _get(dto, "email")

    if not nome or len(nome.strip()) < 2:
        raise ValueError("Nome muito curto")

    return Cliente(
        id=uuid4().hex,
        nome=nome.strip(),
        email=email.strip(),
        status="ativo"
    )

# ❌ ERRADO
def criar_cliente(dto: Any, repo) -> Cliente:  # ERRO: depende de repo
    if repo.existe_email(dto.email):  # ERRO: validação externa
        raise ValueError("Email existe")

    return Cliente(nome=dto.nome)  # ERRO: entidade incompleta
```

🏛️ Domain - PODE e NÃO PODE

✅ PODE:

- Definir entidades como dataclass
- Implementar comportamentos (ativar, desativar)
- Validar invariantes da entidade
- Aplicar regras de negócio puras
- Usar método aplicar_atualizacao_from_any()
- Ter propriedades calculadas

❌ NÃO PODE:

- Importar Pydantic, FastAPI, SQLAlchemy
- Acessar banco de dados
- Ter @staticmethod criar() (só Factory cria)
- Depender de camadas externas
- Fazer validações que precisam de dados externos

```python
# ✅ CORRETO
@dataclass
class Cliente:
    id: str
    nome: str
    email: str
    status: str = "ativo"

    def ativar(self) -> None:
        self.status = "ativo"
        self.updated_at = datetime.utcnow()

    def aplicar_atualizacao_from_any(self, data: Any) -> None:
        novo_nome = _get(data, "nome")
        if novo_nome and len(novo_nome.strip()) >= 2:
            self.nome = novo_nome.strip()

# ❌ ERRADO
@dataclass
class Cliente:
    nome: str

    @staticmethod  # ERRO: só Factory cria
    def criar(nome: str):
        return Cliente(nome=nome)

    def validar_email_unico(self, repo):  # ERRO: depende de repo
        return repo.existe_email(self.email)
```

💾 Repository - PODE e NÃO PODE

✅ PODE:

- Salvar/buscar/atualizar entidades
- Converter Entity ↔ Model (mapeamentos privados)
- Usar SQLAlchemy, MongoDB, etc.
- Fazer queries complexas quando necessário
- Implementar métodos como get_by_email, list, etc.

❌ NÃO PODE:

- Implementar regras de negócio
- Validar dados (isso é do Domain)
- Conhecer detalhes de API ou DTOs
- Fazer cálculos de negócio
- Expor detalhes do banco para outras camadas

```python
# ✅ CORRETO
class ClienteRepository:
    def add(self, cliente: Cliente) -> None:
        model = self._to_model(cliente)
        self.session.add(model)
        self.session.commit()

    def _to_model(self, entity: Cliente) -> ClienteModel:
        return ClienteModel(
            id=entity.id,
            nome=entity.nome,
            email=entity.email
        )

# ❌ ERRADO
class ClienteRepository:
    def add(self, cliente: Cliente) -> None:
        # ERRO: validação (isso é do Domain)
        if len(cliente.nome) < 2:
            raise ValueError("Nome muito curto")

        # ERRO: regra de negócio
        if cliente.idade < 18:
            cliente.status = "menor_idade"
```

---

8. A Árvore da Nossa Arquitetura

Aqui está como você vai organizar seus arquivos, meu querido:

```
projeto/
├── app/
│   ├── __init__.py
│   ├── main.py                    # FastAPI app principal
│   │
│   ├── api/                       # 🌐 Camada API
│   │   ├── __init__.py
│   │   ├── clientes_api.py        # Endpoints de cliente
│   │   ├── funcionarios_api.py    # Endpoints de funcionário
│   │   └── produtos_api.py        # Endpoints de produto
│   │
│   ├── schemas/                   # 📋 Contratos (DTOs)
│   │   ├── __init__.py
│   │   ├── clientes/
│   │   │   ├── __init__.py
│   │   │   ├── requests.py        # ClienteCreateRequest, etc.
│   │   │   └── responses.py       # ClienteResponse, etc.
│   │   ├── funcionarios/
│   │   │   ├── requests.py
│   │   │   └── responses.py
│   │   └── produtos/
│   │       ├── requests.py
│   │       └── responses.py
│   │
│   ├── mappers/                   # 🔄 Tradutores
│   │   ├── __init__.py
│   │   ├── cliente_mapper.py      # to_public(), to_display()
│   │   ├── funcionario_mapper.py
│   │   └── produto_mapper.py
│   │
│   ├── services/                  # ⚙️ Orquestradores
│   │   ├── __init__.py
│   │   ├── cliente_service.py     # criar_cliente_service(), etc.
│   │   ├── funcionario_service.py
│   │   └── produto_service.py
│   │
│   ├── domain/                    # 🏛️ Coração do Sistema
│   │   ├── __init__.py
│   │   ├── clientes/
│   │   │   ├── __init__.py
│   │   │   ├── cliente_entity.py  # Classe Cliente
│   │   │   └── cliente_factory.py # criar_cliente()
│   │   ├── funcionarios/
```

```
│   │   │   ├── funcionario_entity.py
│   │   │   └── funcionario_factory.py
│   │   └── produtos/
│   │       ├── produto_entity.py
│   │       └── produto_factory.py
│   │
│   ├── data/                    # 💾 Persistência
│   │   ├── __init__.py
│   │   ├── models/              # Models do SQLAlchemy
│   │   │   ├── __init__.py
│   │   │   ├── cliente_model.py      # ClienteModel (SQLAlchemy)
│   │   │   ├── funcionario_model.py
│   │   │   └── produto_model.py
│   │   └── repositories/        # Repositórios
│   │       ├── __init__.py
│   │       ├── cliente_repository.py    # ClienteRepository
│   │       ├── funcionario_repository.py
│   │       └── produto_repository.py
│   │
│   ├── integrations/            # 👹 Integrações Externas
│   │   ├── __init__.py
│   │   └── ai/
│   │       ├── __init__.py
│   │       ├── clientes/
│   │       │   ├── __init__.py
│   │       │   ├── payloads.py      # DTOs para IA
│   │       │   └── client.py        # Cliente IA
│   │       ├── funcionarios/
│   │       │   ├── payloads.py
│   │       │   └── client.py
│   │       └── produtos/
│   │           ├── payloads.py
│   │           └── client.py
│   │
│   └── config/                  # ⚙️ Configurações
│       ├── __init__.py
│       ├── database.py          # Configuração do banco
│       └── settings.py          # Configurações gerais
│
├── tests/                       # 🧪 Testes
│   ├── unit/                    # Testes unitários
│   │   ├── test_cliente_entity.py
│   │   ├── test_cliente_factory.py
│   │   └── test_cliente_service.py
```

```
│   └── integration/              # Testes de integração
│       ├── test_cliente_api.py
│       └── test_cliente_repository.py
│
├── migrations/                   # 📝 Migrações do banco
├── requirements.txt              # 📦 Dependências
├── .env                          # 🔐 Variáveis de ambiente
└── README.md                     # 📖 Documentação
```

Explicação dos Diretórios:

📁 **api/**: Todos os endpoints da sua API. Um arquivo por entidade.

📁 **schemas/**: Contratos de entrada e saída. Separados por entidade, com requests e responses.

📁 **mappers/**: Conversores de Entity para Response. Um arquivo por entidade.

📁 **services/**: Orquestradores de casos de uso. Coordenam o fluxo sem implementar regras.

📁 **domain/**: O coração do sistema. Entidades + Factories. Separado por contexto de negócio.

📁 **data/**: Tudo relacionado a persistência. Models (SQLAlchemy) e Repositories.

📁 **integrations/**: Integrações com sistemas externos (IA, APIs de terceiros).

📁 **config/**: Configurações de banco, variáveis de ambiente, etc.

---

9. Exemplo Prático: Cliente

Agora vou te mostrar um exemplo completo, meu querido. Vamos criar um cliente do zero seguindo nossa arquitetura.

9.1 Começando pelo Domain (Coração)

app/domain/clientes/cliente_entity.py

python

```python
from dataclasses import dataclass, field
from datetime import datetime, date
from typing import Any, Optional

def _get(data: Any, name: str, default=None):
    """Helper universal para extrair dados de qualquer fonte"""
    if isinstance(data, dict):
        return data.get(name, default)
    return getattr(data, name, default)

@dataclass
class Cliente:
    """
    Entidade Cliente - representa um cliente no nosso domínio.
    Contém apenas regras de negócio puras.
    """
    id: str
    nome: str
    email: str
    telefone: Optional[str] = None
    data_nascimento: Optional[date] = None
    status: str = "ativo"
    created_at: datetime = field(default_factory=datetime.utcnow)
    updated_at: datetime = field(default_factory=datetime.utcnow)

    def ativar(self) -> None:
        """Ativa o cliente - regra de negócio pura"""
        if self.status == "banido":
            raise ValueError("Cliente banido não pode ser ativado")
        self.status = "ativo"
        self.updated_at = datetime.utcnow()

    def desativar(self) -> None:
        """Desativa o cliente"""
        self.status = "inativo"
        self.updated_at = datetime.utcnow()

    def banir(self, motivo: str) -> None:
        """Bane o cliente por motivo específico"""
        if not motivo:
            raise ValueError("Motivo do banimento é obrigatório")
        self.status = "banido"
        self.updated_at = datetime.utcnow()
```

```python
    def pode_fazer_compras(self) -> bool:
        """Regra de negócio: quando cliente pode comprar"""
        return self.status == "ativo"

    def aplicar_atualizacao_from_any(self, data: Any) -> None:
        """
        Aplica atualizações vindas de qualquer fonte (DTO, dict, etc.)
        Entity sabe como se atualizar
        """
        novo_nome = _get(data, "nome")
        novo_tel = _get(data, "telefone")
        nova_data = _get(data, "data_nascimento") or _get(data, "dataNascimento")
        novo_status = _get(data, "status")

        if novo_nome is not None:
            if not novo_nome or len(novo_nome.strip()) < 2:
                raise ValueError("Nome deve ter pelo menos 2 caracteres")
            self.nome = novo_nome.strip()

        if novo_tel is not None:
            self.telefone = novo_tel

        if nova_data is not None:
            self.data_nascimento = nova_data

        if novo_status == "ativo":
            self.ativar()
        elif novo_status == "inativo":
            self.desativar()

        self.updated_at = datetime.utcnow()
```

app/domain/clientes/cliente_factory.py

```python
```

```python
from typing import Any
from datetime import datetime
from uuid import uuid4
from .cliente_entity import Cliente, _get

class ClienteFactory:
    """
    Factory para criação de entidades Cliente.
    ÚNICA porta de entrada para criar clientes.
    """

    @staticmethod
    def make_client(dto: Any) -> Cliente:
        """
        Cria entidade Cliente completa a partir do DTO.
        Centraliza todas as validações e lógica de criação.
        """
        nome = _get(dto, "nome")
        email = _get(dto, "email")
        telefone = _get(dto, "telefone")
        data_nascimento = _get(dto, "data_nascimento") or _get(dto, "dataNascimento")

        # Validações de criação
        if not nome or len(nome.strip()) < 2:
            raise ValueError("Nome deve ter pelo menos 2 caracteres")
        if not email or not email.strip():
            raise ValueError("Email é obrigatório")

        # Cria entidade completa
        return Cliente(
            id=uuid4().hex,
            nome=nome.strip(),
            email=email.strip(),
            telefone=telefone,
            data_nascimento=data_nascimento,
            status="ativo",
            created_at=datetime.utcnow(),
            updated_at=datetime.utcnow(),
        )

    @staticmethod
    def email_from(dto: Any) -> str | None:
        """Helper para Service extrair email sem acoplamento"""
```

```
        return _get(dto, "email")

    @staticmethod
    def id_from(dto: Any) -> str | None:
        """Helper para Service extrair ID sem acoplamento"""
        return _get(dto, "id")
```

9.2 Definindo os Contratos (Schemas)

app/schemas/clientes/requests.py

python

```python
from pydantic import BaseModel, EmailStr, constr, Field
from typing import Optional
from datetime import date

class ClienteCreateRequest(BaseModel):
    """Contrato para criar cliente"""
    nome: constr(min_length=2, max_length=255) = Field(
        description="Nome completo do cliente"
    )
    email: EmailStr = Field(
        description="Email válido do cliente"
    )
    telefone: Optional[constr(max_length=20)] = Field(
        None, description="Telefone de contato (opcional)"
    )
    data_nascimento: Optional[date] = Field(
        None, description="Data de nascimento (opcional)"
    )

class ClienteUpdateRequest(BaseModel):
    """Contrato para atualizar cliente"""
    nome: Optional[constr(min_length=2, max_length=255)] = None
    telefone: Optional[constr(max_length=20)] = None
    data_nascimento: Optional[date] = None
    status: Optional[str] = Field(
        None, description="Status: ativo, inativo ou banido"
    )

class ClienteSearchRequest(BaseModel):
    """Contrato para buscar clientes"""
    status: Optional[str] = None
    page: int = Field(1, ge=1, description="Página (começando em 1)")
    page_size: int = Field(20, ge=1, le=100, description="Itens por página")
```

app/schemas/clientes/responses.py

```python
```

```python
from pydantic import BaseModel, EmailStr, Field
from typing import List, Optional
from datetime import datetime, date

class ClientePublicResponse(BaseModel):
    """Resposta completa do cliente"""
    id: str
    nome: str
    email: EmailStr
    status: str
    telefone: Optional[str] = None
    data_nascimento: Optional[date] = None
    created_at: datetime
    updated_at: datetime

class ClienteDisplayResponse(BaseModel):
    """Resposta simplificada para telas básicas"""
    nome: str = Field(description="Nome do cliente")
    email: str = Field(description="Email do cliente")

class ClienteListResponse(BaseModel):
    """Resposta para listagem paginada"""
    total: int = Field(description="Total de registros")
    items: List[ClientePublicResponse] = Field(description="Lista de clientes")
```

9.3 Criando o Service (Orquestrador)

app/services/cliente_service.py

python

```python
from app.schemas.clientes.requests import (
    ClienteCreateRequest,
    ClienteUpdateRequest,
    ClienteSearchRequest
)
from app.domain.clientes.cliente_entity import Cliente
from app.domain.clientes.cliente_factory import ClienteFactory
from app.data.repositories.cliente_repository import ClienteRepository

class ClienteService:
    """Service para casos de uso de Cliente"""

    def __init__(self, repo: ClienteRepository):
        self.repo = repo

    def criar_cliente(self, dto: ClienteCreateRequest) -> Cliente:
        """
        Caso de uso: Criar cliente
        Service orquestra, não implementa
        """
        # 1. Validação que depende de dados externos
        email = ClienteFactory.email_from(dto)  # Helper da Factory
        if email and self.repo.get_by_email(email):
            raise ValueError("Email já cadastrado no sistema")

        # 2. Cria entidade via Factory (única porta)
        cliente = ClienteFactory.make_client(dto)

        # 3. Persiste via Repository
        self.repo.add(cliente)

        return cliente

    def atualizar_cliente(self, cliente_id: str, dto: ClienteUpdateRequest) -> Cliente:
        """Caso de uso: Atualizar cliente"""
        # 1. Busca cliente existente
        cliente = self.repo.get_by_id(cliente_id)
        if not cliente:
            raise ValueError("Cliente não encontrado")

        # 2. Aplica mudanças (Entity sabe como fazer)
        cliente.aplicar_atualizacao_from_any(dto)
```

```
        # 3. Persiste mudanças
        self.repo.update(cliente)

        return cliente

    def buscar_cliente(self, cliente_id: str) -> Cliente:
        """Caso de uso: Buscar cliente por ID"""
        cliente = self.repo.get_by_id(cliente_id)
        if not cliente:
            raise ValueError("Cliente não encontrado")
        return cliente

    def buscar_por_email(self, email: str) -> Cliente:
        """Caso de uso: Buscar cliente por email"""
        cliente = self.repo.get_by_email(email)
        if not cliente:
            raise ValueError("Cliente não encontrado")
        return cliente

    def listar_clientes(self, dto: ClienteSearchRequest) -> tuple[list[Cliente], int]:
        """Caso de uso: Listar clientes com filtros e paginação"""
        skip = (dto.page - 1) * dto.page_size
        return self.repo.list(status=dto.status, skip=skip, limit=dto.page_size)

    def banir_cliente(self, cliente_id: str, motivo: str) -> Cliente:
        """Caso de uso: Banir cliente"""
        cliente = self.repo.get_by_id(cliente_id)
        if not cliente:
            raise ValueError("Cliente não encontrado")

        cliente.banir(motivo)  # Entity implementa a regra
        self.repo.update(cliente)

        return cliente
```

9.4 Implementando os Mappers (Tradutores)

app/mappers/cliente_mapper.py

python

```python
from app.domain.clientes.cliente_entity import Cliente
from app.schemas.clientes.responses import (
    ClientePublicResponse,
    ClienteDisplayResponse
)

class ClienteMapper:
    """Mapper para conversões de Cliente"""

    @staticmethod
    def to_public(c: Cliente) -> ClientePublicResponse:
        """Converte Entity para resposta completa"""
        return ClientePublicResponse(
            id=c.id,
            nome=c.nome,
            email=c.email,
            status=c.status,
            telefone=c.telefone,
            data_nascimento=c.data_nascimento,
            created_at=c.created_at,
            updated_at=c.updated_at,
        )

    @staticmethod
    def to_display(c: Cliente) -> ClienteDisplayResponse:
        """Converte Entity para resposta simplificada"""
        return ClienteDisplayResponse(
            nome=c.nome,
            email=c.email
        )
```

9.5 Criando a Persistência (Repository)

app/data/models/cliente_model.py

```python
```

```python
from sqlalchemy import Column, String, DateTime, Date
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class ClienteModel(Base):
    """Model SQLAlchemy para tabela clientes"""
    __tablename__ = "clientes"

    id = Column(String, primary_key=True)
    nome = Column(String(255), nullable=False)
    email = Column(String(255), unique=True, nullable=False, index=True)
    telefone = Column(String(20))
    data_nascimento = Column(Date)
    status = Column(String(20), default="ativo", index=True)
    created_at = Column(DateTime, nullable=False)
    updated_at = Column(DateTime, nullable=False)
```

app/data/repositories/cliente_repository.py

```python
```

```python
from sqlalchemy.orm import Session
from typing import Optional, List, Tuple
from app.domain.clientes.cliente_entity import Cliente
from ..models.cliente_model import ClienteModel

class ClienteRepository:
    """Repository para persistência de clientes"""

    def __init__(self, session: Session):
        self.session = session

    def add(self, cliente: Cliente) -> None:
        """Adiciona novo cliente"""
        model = self._to_model(cliente)
        self.session.add(model)
        self.session.commit()

    def update(self, cliente: Cliente) -> None:
        """Atualiza cliente existente"""
        model = self._to_model(cliente)
        self.session.merge(model)
        self.session.commit()

    def delete(self, cliente_id: str) -> None:
        """Remove cliente"""
        self.session.query(ClienteModel).filter_by(id=cliente_id).delete()
        self.session.commit()

    def get_by_id(self, cliente_id: str) -> Optional[Cliente]:
        """Busca cliente por ID"""
        model = self.session.query(ClienteModel).filter_by(id=cliente_id).first()
        return self._to_entity(model) if model else None

    def get_by_email(self, email: str) -> Optional[Cliente]:
        """Busca cliente por email"""
        model = self.session.query(ClienteModel).filter_by(email=email).first()
        return self._to_entity(model) if model else None

    def list(self, status: Optional[str] = None, skip: int = 0,
             limit: int = 20) -> Tuple[List[Cliente], int]:
        """Lista clientes com filtros e paginação"""
        query = self.session.query(ClienteModel)
```

```python
        if status:
            query = query.filter(ClienteModel.status == status)

        total = query.count()
        items = query.offset(skip).limit(limit).all()

        return [self._to_entity(item) for item in items], total

    def _to_entity(self, model: ClienteModel) -> Cliente:
        """Converte Model do banco para Entity do domínio"""
        return Cliente(
            id=model.id,
            nome=model.nome,
            email=model.email,
            telefone=model.telefone,
            data_nascimento=model.data_nascimento,
            status=model.status,
            created_at=model.created_at,
            updated_at=model.updated_at,
        )

    def _to_model(self, entity: Cliente) -> ClienteModel:
        """Converte Entity do domínio para Model do banco"""
        return ClienteModel(
            id=entity.id,
            nome=entity.nome,
            email=entity.email,
            telefone=entity.telefone,
            data_nascimento=entity.data_nascimento,
            status=entity.status,
            created_at=entity.created_at,
            updated_at=entity.updated_at,
        )
```

9.6 Criando a API (Interface Externa)

app/api/clientes_api.py

python

```python
from fastapi import APIRouter, Depends, HTTPException
from typing import List
from app.schemas.clientes.requests import (
    ClienteCreateRequest,
    ClienteUpdateRequest,
    ClienteSearchRequest
)
from app.schemas.clientes.responses import (
    ClientePublicResponse,
    ClienteDisplayResponse,
    ClienteListResponse
)
from app.services.cliente_service import ClienteService
from app.mappers.cliente_mapper import ClienteMapper
from app.data.repositories.cliente_repository import ClienteRepository

router = APIRouter(prefix="/clientes", tags=["clientes"])

def get_cliente_service():
    """Dependency injection do Service com Repository"""
    repo = ClienteRepository(session)
    return ClienteService(repo)

@router.post("", response_model=ClientePublicResponse, status_code=201)
def create(dto: ClienteCreateRequest, repo=Depends(repo_dep)):
    """Cria novo cliente"""
    try:
        entity = ClienteService.criar_cliente(dto, repo)
        return ClienteMapper.to_public(entity)
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))

@router.put("/{cliente_id}", response_model=ClientePublicResponse)
def update(cliente_id: str, dto: ClienteUpdateRequest, repo=Depends(repo_dep)):
    """Atualiza cliente existente"""
    try:
        entity = ClienteService.atualizar_cliente(cliente_id, dto, repo)
        return ClienteMapper.to_public(entity)
    except ValueError as e:
        raise HTTPException(status_code=404, detail=str(e))

@router.get("/{cliente_id}", response_model=ClientePublicResponse)
def get(cliente_id: str, repo=Depends(repo_dep)):
```

```
    """Busca cliente por ID - resposta completa"""
    try:
        entity = ClienteService.buscar_cliente(cliente_id, repo)
        return ClienteMapper.to_public(entity)
    except ValueError as e:
        raise HTTPException(status_code=404, detail=str(e))

@router.get("/{cliente_id}/display", response_model=ClienteDisplayResponse)
def get_display(cliente_id: str, repo=Depends(repo_dep)):
    """Busca cliente por ID - resposta simplificada para telas básicas"""
    try:
        entity = ClienteService.buscar_cliente(cliente_id, repo)
        return ClienteMapper.to_display(entity)
    except ValueError as e:
        raise HTTPException(status_code=404, detail=str(e))

@router.get("/by-email/{email}", response_model=ClientePublicResponse)
def get_by_email(email: str, repo=Depends(repo_dep)):
    """Busca cliente por email"""
    try:
        entity = ClienteService.buscar_por_email(email, repo)
        return ClienteMapper.to_public(entity)
    except ValueError as e:
        raise HTTPException(status_code=404, detail=str(e))

@router.get("", response_model=ClienteListResponse)
def list_clientes(
    status: str = None,
    page: int = 1,
    page_size: int = 20,
    repo=Depends(repo_dep)
):
    """Lista clientes com filtros e paginação"""
    items, total = ClienteService.listar_clientes(
        ClienteSearchRequest(status=status, page=page, page_size=page_size),
        repo
    )
    return ClienteListResponse(
        total=total,
        items=[ClienteMapper.to_public(c) for c in items]
    )

@router.get("/display/lista", response_model=List[ClienteDisplayResponse])
def list_display(repo=Depends(repo_dep)):
```

```
    """Lista clientes simplificada para telas básicas"""
    items, _ = ClienteService.listar_clientes(ClienteSearchRequest(), repo)
    return [ClienteMapper.to_display(c) for c in items]

@router.post("/{cliente_id}/banir", response_model=ClientePublicResponse)
def ban(cliente_id: str, motivo: str, repo=Depends(repo_dep)):
    """Bane cliente do sistema"""
    try:
        entity = ClienteService.banir_cliente(cliente_id, motivo, repo)
        return ClienteMapper.to_public(entity)
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))
```

9.7 Adicionando Integração com IA

app/integrations/ai/clientes/payloads.py

python

```
from pydantic import BaseModel, EmailStr, Field
from typing import Optional, List
from datetime import date

class ClienteAIPayload(BaseModel):
    """Payload para análise de cliente via IA"""
    nome: str = Field(description="Nome completo do cliente")
    email: EmailStr = Field(description="Email do cliente")
    telefone: Optional[str] = Field(None, description="Telefone de contato")
    data_nascimento: Optional[date] = Field(None, description="Data de nascimento")
    historico_compras: Optional[List[dict]] = Field(None, description="Histórico de compras")

class ClienteAIResponse(BaseModel):
    """Resposta da análise de IA"""
    id: str = Field(description="ID da análise")
    score_engajamento: float = Field(description="Score de engajamento (0-1)")
    categoria_cliente: str = Field(description="Categoria do cliente")
    recomendacoes: List[str] = Field(description="Recomendações personalizadas")
    proximo_contato: Optional[str] = Field(None, description="Melhor momento para contato")
```

app/integrations/ai/clientes/client.py

python

```python
from .payloads import ClienteAIPayload, ClienteAIResponse

class ClienteAIClient:
    """Cliente para integração com serviços de IA"""

    def __init__(self, api_key: str = None):
        self.api_key = api_key

    def analisar_perfil(self, payload: ClienteAIPayload) -> ClienteAIResponse:
        """
        Analisa perfil do cliente usando IA
        Por enquanto é um mock, mas pode integrar com serviços reais
        """
        # Mock de análise
        score = 0.75 if payload.historico_compras else 0.4
        categoria = "premium" if score > 0.7 else "regular"

        return ClienteAIResponse(
            id="ai_analysis_123",
            score_engajamento=score,
            categoria_cliente=categoria,
            recomendacoes=[
                "Oferecer desconto em produtos premium",
                "Enviar newsletter personalizada",
                "Agendar ligação comercial"
            ],
            proximo_contato="manhã"
        )

    def calcular_score_engajamento(self, historico: List[dict]) -> float:
        """Calcula score baseado no histórico de compras"""
        if not historico:
            return 0.0

        # Lógica simplificada de cálculo
        total_compras = len(historico)
        return min(total_compras * 0.1, 1.0)
```

---

10. Fluxo Completo na Prática

Agora vou te mostrar como tudo funciona junto, meu querido. É como ver toda a orquestra tocando

em harmonia.

10.1 Cenário: Criando um Cliente

1. Request chega na API:

```http
POST /clientes
Content-Type: application/json

{
    "nome": "João Silva",
    "email": "joao@email.com",
    "telefone": "11999999999",
    "data_nascimento": "1990-05-15"
}
```

2. FastAPI converte para DTO automaticamente:

```python
# Pydantic faz isso automaticamente
dto = ClienteCreateRequest(
    nome="João Silva",
    email="joao@email.com",
    telefone="11999999999",
    data_nascimento=date(1990, 5, 15)
)
```

3. API chama Service:

```python
@router.post("/clientes")
def create(dto: ClienteCreateRequest, repo=Depends(repo_dep)):
    entity = criar_cliente_service(dto, repo)  # ← Chama Service
    return to_public(entity)  # ← Usa Mapper
```

4. Service orquestra o processo:

```python
```

```python
def criar_cliente_service(dto: ClienteCreateRequest, repo) -> Cliente:
    # Validação que depende de dados externos
    email = email_from(dto)  # ← Helper da Factory
    if repo.get_by_email(email):
        raise ValueError("Email já existe")

    # Cria entidade via Factory
    cliente = criar_cliente(dto)  # ← Factory única

    # Persiste
    repo.add(cliente)  # ← Repository

    return cliente
```

5. Service cria entidade via Factory:

python

```python
def criar_cliente_service(dto: ClienteCreateRequest, repo) -> Cliente:
    email = ClienteFactory.email_from(dto)  # ← Helper da Factory
    if repo.get_by_email(email):
        raise ValueError("Email já existe")

    # Cria entidade via Factory
    cliente = ClienteFactory.make_client(dto)  # ← Factory única

    # Persiste
    repo.add(cliente)  # ← Repository

    return cliente
```

6. Factory cria a entidade:

python

```python
class ClienteFactory:
    @staticmethod
    def make_client(dto: Any) -> Cliente:
        nome = _get(dto, "nome")  # ← Extrai dados
        email = _get(dto, "email")
        # ... validações ...

        return Cliente(  # ← Cria entidade completa
            id=uuid4().hex,
            nome=nome.strip(),
            email=email.strip(),
            status="ativo",
            created_at=datetime.utcnow(),
            updated_at=datetime.utcnow()
        )
```

6. Repository persiste no banco:

```python
def add(self, cliente: Cliente) -> None:
    model = self._to_model(cliente)  # ← Converte Entity→Model
    self.session.add(model)
    self.session.commit()
```

7. Mapper converte resposta:

```python
def to_public(c: Cliente) -> ClientePublicResponse:
    return ClientePublicResponse(  # ← Entity→DTO
        id=c.id,
        nome=c.nome,
        email=c.email,
        status=c.status,
        created_at=c.created_at,
        updated_at=c.updated_at
    )
```

8. API retorna resposta:

```json
```

```
{
    "id": "a1b2c3d4",
    "nome": "João Silva",
    "email": "joao@email.com",
    "telefone": "11999999999",
    "data_nascimento": "1990-05-15",
    "status": "ativo",
    "created_at": "2024-01-15T10:30:00",
    "updated_at": "2024-01-15T10:30:00"
}
```

10.2 Cenário: Atualizando um Cliente

Request:

```http
PUT /clientes/a1b2c3d4
{
    "nome": "João Santos Silva",
    "telefone": "11888888888"
}
```

Fluxo interno:

1. API → Service (`atualizar_cliente`)
2. Service → Repository (`get_by_id`)
3. Repository → retorna Entity
4. Service → Entity (`aplicar_atualizacao_from_any`)
5. Entity → aplica mudanças com validações
6. Service → Repository (`update`)
7. Repository → persiste mudanças
8. Service → retorna Entity atualizada
9. API → Mapper (`to_public`)
10. API → retorna resposta

10.3 Cenário: Listagem Simplificada

Request:

```http
GET /clientes/display/lista
```

Diferencial:

- Usa o mesmo Service (`listar_clientes`)
- Usa Mapper diferente (`to_display` em vez de `to_public`)
- Retorna apenas nome e email

Response:

```json
[
  {
    "nome": "João Santos Silva",
    "email": "joao@email.com"
  },
  {
    "nome": "Maria Oliveira",
    "email": "maria@email.com"
  }
]
```

---

11. Erros Comuns e Como Evitar

Meu querido, aqui estão os principais erros que vejo estudantes cometendo. Preste atenção para não cair nessas armadilhas!

❌ ERRO 1: Service acessando campos do DTO

ERRADO:

```python
def criar_cliente(dto: ClienteCreateRequest, repo) -> Cliente:
    if repo.existe_email(dto.email):  # ← ERRO: dto.email
        raise ValueError("Email existe")

    cliente = Cliente(nome=dto.nome, email=dto.email)  # ← ERRO: direto
```

CORRETO:

```python
def criar_cliente(dto: ClienteCreateRequest, repo) -> Cliente:
    email = email_from(dto)  # ← Helper da Factory
    if repo.existe_email(email):
        raise ValueError("Email existe")

    cliente = criar_cliente(dto)  # ← Factory única
```

❌ ERRO 2: Entity com método estático de criação

ERRADO:

```python
@dataclass
class Cliente:
    nome: str

    @staticmethod
    def criar(nome: str) -> "Cliente":  # ← ERRO: duplica Factory
        return Cliente(nome=nome)
```

CORRETO:

```python
@dataclass
class Cliente:
    nome: str

    def ativar(self) -> None:  # ← Só comportamentos
        self.status = "ativo"

# Factory separada
def criar_cliente(dto: Any) -> Cliente:  # ← Única porta
    return Cliente(nome=_get(dto, "nome"))
```

❌ ERRO 3: API retornando Entity diretamente

ERRADO:

```python
@router.get("/clientes/{id}")
def get(cliente_id: str, repo=Depends(repo_dep)):
    return ClienteService.buscar_cliente(cliente_id, repo)  # ← ERRO: Entity crua
```

CORRETO:

```python
@router.get("/clientes/{id}")
def get(cliente_id: str, repo=Depends(repo_dep)):
    entity = ClienteService.buscar_cliente(cliente_id, repo)
    return ClienteMapper.to_public(entity)  # ← Mapper converte
```

❌ ERRO 4: Repository com regras de negócio

ERRADO:

```python
class ClienteRepository:
    def add(self, cliente: Cliente) -> None:
        if len(cliente.nome) < 2:  # ← ERRO: validação aqui
            raise ValueError("Nome muito curto")

        if cliente.idade < 18:  # ← ERRO: regra de negócio
            cliente.status = "menor_idade"
```

CORRETO:

```python
class ClienteRepository:
    def add(self, cliente: Cliente) -> None:
        model = self._to_model(cliente)  # ← Só persistência
```

orajoso! Não se apavore nem desanime, pois o Senhor, o seu Deus, estará c