



ENSINO MÉDIO INTEGRADO - INFORMÁTICA

Disciplina de Desenvolvimento Web | Backend

Aula 08: Framework FastAPI - Arquitetura Limpa

Gil Eduardo de Andrade

Conceitos Preliminares

(<https://blog.cleancoder.com/uncle-bob/2011/11/22/Clean-Architecture.html>)

Introdução à Arquitetura Limpa (*Clean Architecture*)

A Arquitetura Limpa, popularizada por Robert C. Martin (Uncle Bob), é um conjunto de princípios de design de software que visa criar sistemas que são:

- Independentes de Frameworks: A arquitetura não deve depender da existência de alguma biblioteca de software com muitos recursos.
- Testáveis: As regras de negócio podem ser testadas sem a UI, o banco de dados, o servidor web ou qualquer outro elemento externo.
- Independentes de UI: A UI pode mudar facilmente sem alterar o resto do sistema.
- Independentes de Banco de Dados: Você pode trocar o banco de dados (SQL, NoSQL, etc.) sem alterar as regras de negócio.
- Independentes de Agentes Externos: As regras de negócio não devem saber nada sobre o mundo externo.

O cerne da Arquitetura Limpa é a **separação de preocupações** em camadas concêntricas, onde as dependências sempre apontam para dentro. Isso significa que as camadas mais internas (regras de negócio) não devem ter conhecimento das camadas mais externas (frameworks, bancos de dados, UI). Essa estrutura é frequentemente visualizada como cebola, com o domínio no centro e as tecnologias externas nas camadas mais externas.

Camadas Principais

1. Entidades (*Entities*): regras de negócio mais gerais e de alto nível. São objetos que encapsulam dados e o comportamento mais fundamental da aplicação. Elas não devem depender de nada externo.

2. Casos de Uso (Use Cases): contêm a lógica de negócio específica da aplicação. Orquestram o fluxo de dados para e das entidades, e direcionam as entidades para usar suas regras de negócio. Eles dependem apenas das Entidades.
3. Adaptadores de Interface (Interface Adapters): convertem dados do formato mais conveniente para os Casos de Uso e Entidades para o formato mais conveniente para alguma agência externa (e vice-versa). Exemplos incluem Gateways de Banco de Dados (*Repositories*), *Presenters* e Controladores (*Endpoints de API*).
4. Frameworks e Drivers: camada mais externa, composta por frameworks web (FastAPI), bancos de dados (PostgreSQL), ferramentas, etc. Essa camada contém os detalhes de implementação que são mais propensos a mudar.

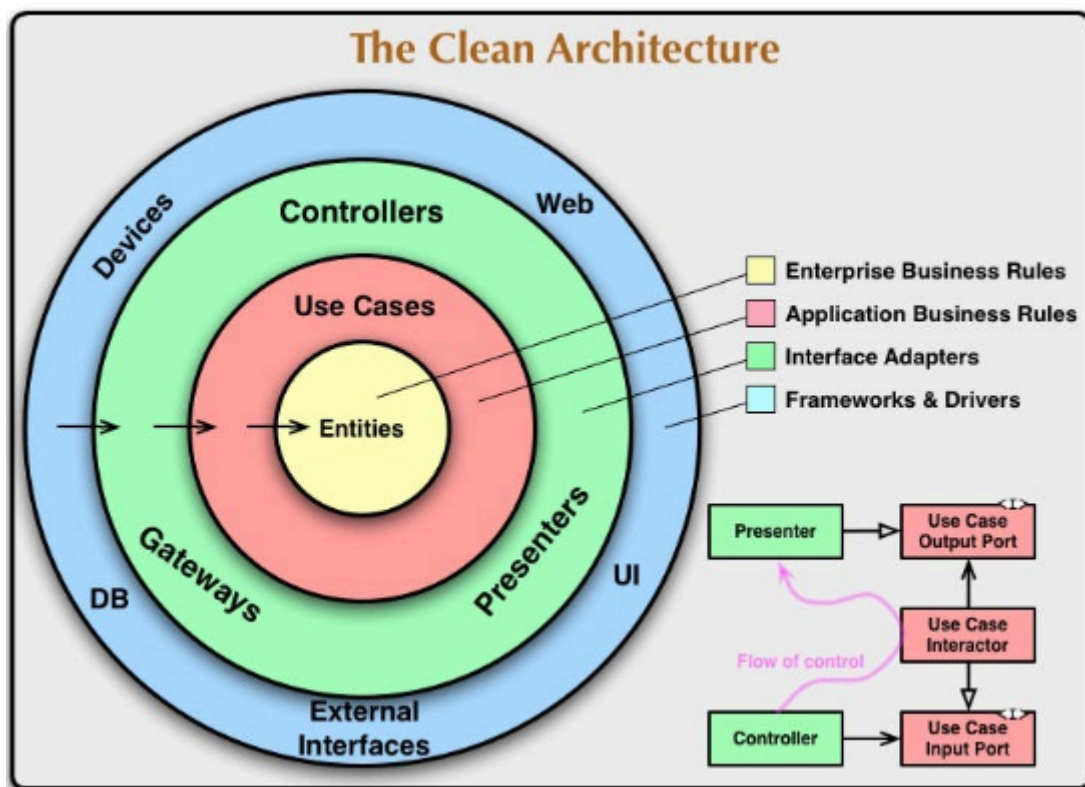


Figura 01: Representação da Arquitetura Limpa (*Clean Architecture*)

Fonte: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

OBS.: as regras de negócio da empresa são as políticas, leis internas e diretrizes que ditam como a organização funciona, sendo independentes da tecnologia - elas existem mesmo se a empresa usar papel e caneta. As regras de negócio da

aplicação são a tradução dessas políticas para o software, controlando a interface, o fluxo de telas, a segurança e a forma como os dados técnicos são validados.

Exemplo:

Regras de Negócio da Empresa

1. Clientes do plano Premium têm frete grátis em compras acima de R\$ 200,00.
2. Nenhum funcionário pode aprovar o próprio reembolso de despesas.
3. **Foco:** Valores da empresa, conformidade fiscal e modelo de operação.

Regras de Negócio da Aplicação

1. O sistema deve exibir uma tag verde "Frete Grátis" na página do carrinho e desabilitar as opções de entrega paga quando o total atingir R\$ 200,00.
2. O botão "Aprovar" deve ficar desabilitado caso o usuário logado seja o mesmo solicitante da despesa, impedindo o clique.
3. **Foco:** UX (Experiência do Usuário), segurança e integridade técnica do software;

Análise da Estrutura do Projeto Atual

A estrutura atual do projeto é a seguinte:

Python

```
|— app
| |— alembic
| |— core
| | |— _init_.py
| | |— roles.py
| | |— security.py
| | |— users.py
| |— endpoints
| | |— _init_.py
| | |— auth.py
| | |— curso.py
| | |— disciplina.py
| |— models
| | |— _init_.py
| | |— curso.py
| | |— disciplina.py
```

```
| | └─ user.py
| └─ schemas
| | └─ _init_.py
| | └─ curso_schema.py
| | └─ disciplina_schema.py
| | └─ user_schema.py
| └─ _init_.py
| └─ database.py
| └─ main.py
└─ Dockerfile
└─ alembic.ini
└─ docker-compose.yml
└─ requirements.txt
```

Pontos Fortes:

- Organização Inicial: o projeto já possui uma boa separação de módulos (endpoints, models, schemas, core), o que facilita a refatoração.
- Uso de FastAPI: o framework FastAPI é moderno e performático, ideal para construção de APIs.
- Dockerização: o uso de Dockerfile e docker-compose.yml demonstra uma preocupação com a padronização do ambiente de desenvolvimento e deploy.
- Autenticação e Autorização: a implementação com *fastapi-users* e *SQLAlchemyUserDatabase* já está funcional, o que é um bom ponto de partida.
- SQLAlchemy: utilização de um ORM robusto para interação com o banco de dados.

Pontos de Acoplamento e Oportunidades de Melhoria (Ótica da Clean Architecture)

Atualmente, existe um acoplamento significativo entre as camadas, o que é comum em projetos que ainda não aplicam a Arquitetura Limpa. Os principais pontos são:

- Endpoints diretamente acoplados ao ORM (SQLAlchemy): nos arquivos `app/endpoints/curso.py` e `app/endpoints/disciplina.py`, a lógica de negócio (criação, listagem, atualização, exclusão) e a interação com o banco de dados (via `db.query(Curso).filter( . )`) estão diretamente nos endpoints. Isso

dificulta a testabilidade da lógica de negócio e a troca do ORM ou do banco de dados.

- Modelos de ORM como Entidades de Domínio: os modelos definidos em **`app/models/curso.py`**, **`app/models/disciplina.py`** e **`app/models/user.py`** são classes do SQLAlchemy (`declarative_base`). Idealmente, as entidades de domínio (as regras de negócio puras) não deveriam ter dependência de um ORM específico.
- Lógica de Negócio Espalhada: partes da lógica de negócio estão misturadas nos endpoints, dificultando a reutilização e a manutenção.
- Dependência de `app.database`: a função **`get_db()`** é injetada diretamente nos endpoints, acoplando-os à implementação específica do SQLAlchemy e à forma como a sessão do banco é obtida.
- `fastapi-users` e `SQLAlchemyUserDatabase`: embora sejam ferramentas excelentes, a integração direta no **`app/core/users.py`** e **`app/endpoints/auth.py`** pode ser refatorada para que as regras de negócio de autenticação sejam mais independentes da implementação do `fastapi-users`.

O objetivo da refatoração será desacoplar essas camadas, tornando o sistema mais flexível, testável e fácil de manter, sem perder as vantagens do FastAPI, Docker e SQLAlchemy.

Proposta: Nova Estrutura de Pastas com *Clean Architecture*

Para aplicar a Arquitetura Limpa, vamos utilizar uma nova estrutura de pastas que reflete as camadas concêntricas da *Clean Architecture*, garantindo que as dependências fluam apenas para dentro. A ideia é isolar as regras de negócio (domínio) de detalhes de infraestrutura e do framework web.

Python

```
|— app
| |— core                # Configurações globais e utilitários
| | |— _init_.py
| | |— config.py         # Variáveis de ambiente, configurações ge
| | |— exceptions.py     # Exceções customizadas
| |— domain              # Camada mais interna: Entidades e regras
| | |— _init_.py
| | |— entities          # Classes de domínio (Curso, Disciplina, etc)
```



```
| | | └─ _init_.py
| | | └─ aluno.py
| | | └─ curso.py
| | | └─ disciplina.py
| | | └─ matricula.py
| | | └─ user.py
| | └─ repositories      # Interfaces para acesso a dados
| | | └─ _init_.py
| | | └─ aluno_repository.py
| | | └─ curso_repository.py
| | | └─ disciplina_repository.py
| | | └─ matricula_repository.py
| | └─ services          # Serviços de domínio (lógica de negócio)
| | | └─ _init_.py
| | | └─ auth_service.py # Lógica de autenticação e autorização
| └─ application        # Casos de Uso (Application Services)
| | └─ _init_.py
| | └─ use_cases # Lógica de orquestração - funcionalidades
| | | └─ aluno.py
| | | └─ curso.py
| | | └─ disciplina.py
| | | └─ matricula.py
| | └─ dtos              # Data Transfer Objects (DTOs)
| | | └─ _init_.py
| | | └─ aluno.py
| | | └─ curso.py
| | | └─ disciplina.py
| | | └─ matricula.py
| └─ infrastructure # Implementações concretas dos repositórios
| | └─ _init_.py
| | └─ database          # Configuração do SQLAlchemy, modelos, etc
| | | └─ _init_.py
| | | └─ models.py        # Modelos do SQLAlchemy (CursoModel, Disc
| | | └─ session.py      # Função get_db()
| | └─ repositories      # Implementações concretas dos repositórios
| | | └─ _init_.py
| | | └─ mappers.py
| | | └─ aluno_repository_impl.py
| | | └─ curso_repository_impl.py
| | | └─ disciplina_repository_impl.py
| | | └─ matricula_repository_impl.py
```



```
| | |─ security          # Implementação da autenticação e autorização
| | |─ _init_.py
| | |─ auth_backend.py
| | |─ password_hasher.py
| | |─ user_manager.py
| |─ presentation      # Camada mais externa: Endpoints FastAPI
| |─ _init_.py
| |─ api              # Routers FastAPI
| | |─ _init_.py
| | |─ auth_router.py
| | |─ aluno_router.py
| | |─ curso_router.py
| | |─ disciplina_router.py
| | |─ matricula_router.py
| |─ dependencies     # Funções de injeção de dependência
| | |─ _init_.py
| |─ schemas          # Schemas Pydantic para validação de tipagem
| | |─ _init_.py
| | |─ aluno_schemas.py
| | |─ curso_schemas.py
| | |─ disciplina_schemas.py
| | |─ matricula_schemas.py
| | |─ user_schemas.py
| |─ main.py          # Ponto de entrada da aplicação FastAPI
|─ Dockerfile
|─ alembic.ini
|─ docker-compose.yml
|─ requirements.txt
|─ tests              # Testes unitários, de integração e end-t
```

OBS.: o *Pydantic* é a principal biblioteca de validação e tipagem de dados em Python. Ela transforma as dicas de tipo (type hints) do Python em regras aplicadas em tempo de execução, garantindo que os dados correspondam à estrutura esperada antes do processamento.

Explicação das Camadas:

1. **app/core**: contém as configurações globais, utilitários e exceções que são usados em várias camadas, mas que não representam regras de negócio do domínio. É uma camada de suporte.
2. **app/domain (Entidades e Regras de Negócio)**:
 - a. **entities**: define as classes de domínio puro (ex: Curso, Disciplina, User). Estas classes devem ser o mais simples possível, sem dependências de frameworks ou ORMs. Podem ser dataclasses ou classes Pydantic básicas, focadas apenas nos atributos e comportamentos essenciais do domínio. Elas representam o coração do seu sistema.
 - b. **repositories**: define as interfaces (classes abstratas) que os Casos de Uso utilizarão para interagir com o armazenamento de dados. Por exemplo, ICursoRepository terá métodos como get_by_id, save, delete. A camada de domínio não sabe como esses métodos são implementados, apenas que eles existem.
 - c. **services**: contém a lógica de negócio que não pertence a uma única entidade, mas orquestra o comportamento de várias entidades ou realiza operações complexas do domínio. Por exemplo, um AuthService que lida com a lógica de autenticação e autorização.
3. **app/application (Casos de Uso)**:
 - a. **use_cases**: cada arquivo representa um caso de uso específico da aplicação (ex: CriarCursoUseCase, ListarDisciplinasUseCase). Eles orquestram o fluxo de dados, chamam as entidades de domínio para aplicar as regras de negócio e utilizam as interfaces de repositório para persistir ou recuperar dados. Não devem conter lógica de framework (FastAPI) ou detalhes de banco de dados (SQLAlchemy).
 - b. **dtos**: define os *Data Transfer Objects* (DTOs) que são usados como entrada e saída dos Casos de Uso. São modelos de dados simples, geralmente *Pydantic*, que representam os dados necessários para uma operação específica, desacoplados dos schemas da API e dos modelos do ORM.
4. **app/infrastructure (Implementações)**:



- a. **database**: contém a configuração do SQLAlchemy, a função `get_db()` e os modelos do ORM (ex: `CursoModel`, `DisciplinaModel`, `UserModel`). Estes são os modelos que interagem diretamente com o banco de dados. Eles são específicos do SQLAlchemy.
- b. **repositories**: contém as implementações concretas das interfaces de repositório definidas na camada de domínio. Por exemplo, `SQLAlchemyCursoRepository` implementará `ICursoRepository` usando os `CursoModel` do SQLAlchemy para interagir com o banco de dados. Esta camada é responsável por mapear as entidades de domínio e os modelos do ORM.
- c. **security**: contém a implementação concreta da autenticação e autorização, utilizando `fastapi-users` e `SQLAlchemyUserDatabase`. Aqui, os detalhes de como a segurança é aplicada são encapsulados.

5. app/presentation (API FastAPI):

- a. **api**: contém os `APIRouter` do *FastAPI*. Estes são os endpoints que recebem as requisições HTTP, validam os dados de entrada (usando Pydantic schemas), invocam os Casos de Uso apropriados e retornam as respostas HTTP. Eles dependem dos Casos de Uso e dos Schemas.
- b. **dependencies**: funções de injeção de dependência do *FastAPI* que fornecem as instâncias dos Casos de Uso e Repositórios para os endpoints. É aqui que as implementações concretas da infraestrutura são injetadas nos endpoints, seguindo o princípio da Inversão de Controle (IoC).
- c. **schemas**: define os *schemas Pydantic* para validação de entrada e saída dos dados da API. Estes *schemas* são específicos da camada de apresentação e podem ser diferentes dos DTOs da camada de aplicação ou das entidades de domínio.

Fluxo de Dependências

As dependências sempre fluem de fora para dentro:

Presentation $\Rightarrow$ Application $\Rightarrow$ Domain (Apresentação $\Rightarrow$ Aplicação $\Rightarrow$ Domínio)
Infrastructure $\Rightarrow$ Domain / Application (Infraestrutura $\Rightarrow$ Domínio / Aplicação)

Isso significa que a camada de Domínio não conhece a camada de Aplicação, Infraestrutura ou Apresentação. A camada de Aplicação conhece a camada de Domínio, mas não conhece as camadas de Infraestrutura ou Apresentação. A camada de Infraestrutura implementa as interfaces definidas na camada de Domínio e Aplicação, e a camada de Apresentação orquestra tudo, utilizando a camada de Aplicação e Infraestrutura.

Simulando Requisições - Entendendo Fluxo de Execução

A seguir é apresentado um diagrama de fluxo que ilustra o caminho percorrido (camadas e arquivos acessados) por duas requisições distintas: uma para Listagem de Cursos (Leitura) e outra para o Cadastro de Disciplinas (Escrita).

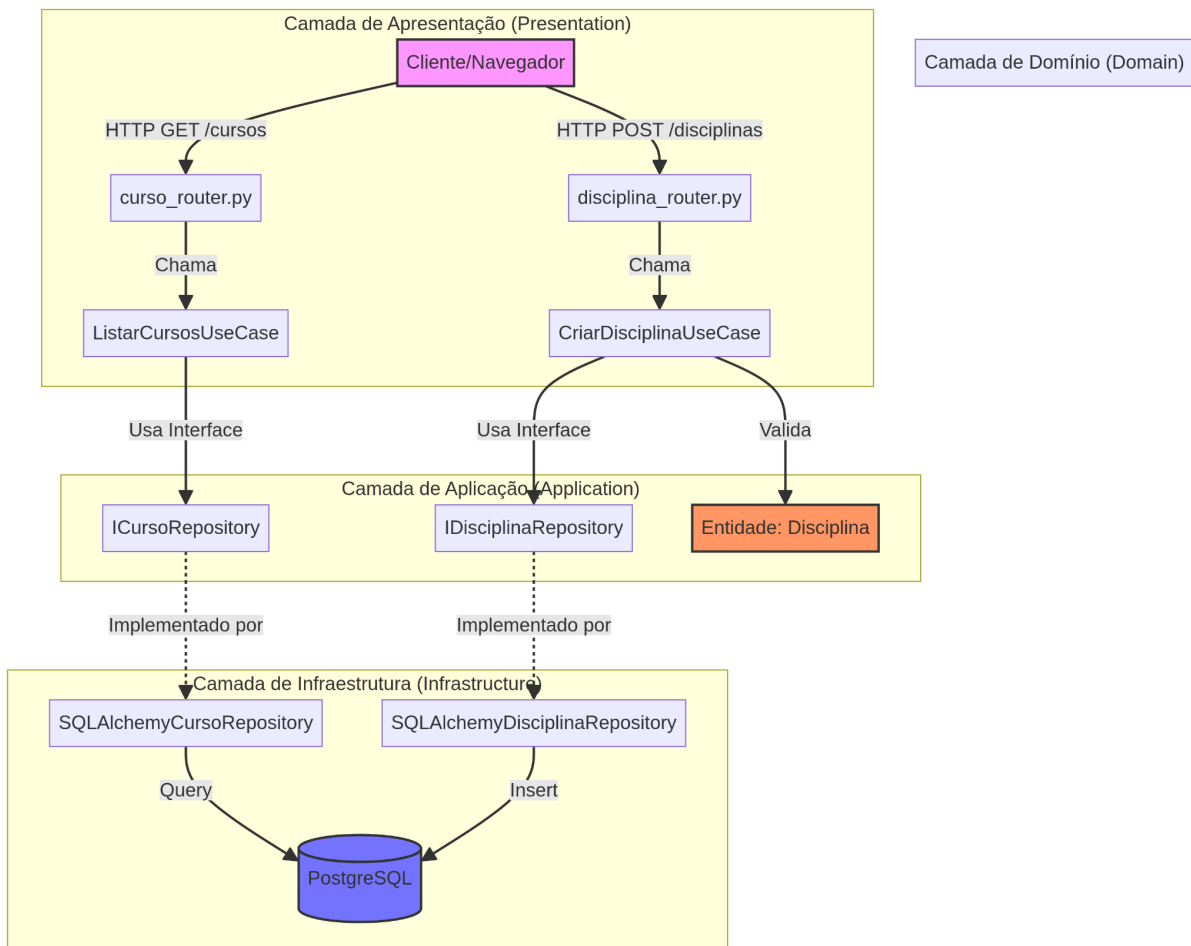


Figura 02: Fluxo de Execução - Requisições Efetuadas a Arquitetura Limpa Projetada.

Detalhamento do Fluxo: Listar Cursos

ETAPA	CAMADA	ARQUIVO UTILIZADO	FUNÇÃO E RESPONSABILIDADE
1. Entrada	Apresentação	<i>presentation/api/curso_router.py</i>	Recebe o GET /cursos, verifica a



			autorização do usuário e chama o Caso de Uso.
2. Orquestração	Aplicação	<i>application/use_cases/curso_use_cases.py</i>	A classe “ <i>ListarCursosUseCase</i> ” solicita os dados ao repositório (via interface).
3. Abstração	Domínio	<i>domain/repositories/curso_repository.py</i>	Define o contrato “ <i>ICursoRepository</i> ”. O Caso de Uso não sabe como os dados são buscados, apenas que podem ser buscados.
4. Execução	Infraestrutura	<i>infrastructure/repositories/curso_repository_impl.py</i>	A classe “ <i>SQLAlchemyCursoRepository</i> ” executa a query real no banco de dados PostgreSQL.
5. Retorno	Apresentação	<i>presentation/schemas/curso_schemas.py</i>	O resultado é convertido em um “ <i>Schema Pydantic</i> ” para ser enviado como JSON ao cliente.

Detalhamento do Fluxo: Criar Disciplina

ETAPA	CAMADA	ARQUIVO UTILIZADO	FUNÇÃO E RESPONSABILIDADE
1. Entrada	Apresentação	<i>presentation/api/disciplina_router.py</i>	Recebe o POST /disciplinas com o JSON de entrada, valida o formato e chama o Caso de Uso.
2. Regra de Negócio	Aplicação	<i>application/use_cases/disciplina_use_cases.py</i>	“ <i>CriarDisciplinaUseCase</i> ” verifica se o “ <i>curso_id</i> ” informado existe antes de prosseguir.
3. Criação	Domínio	<i>domain/entities/disciplina.py</i>	Cria uma instância da entidade Disciplina. Aqui são aplicadas regras puras (ex: carga horária não pode ser negativa).
4. Persistência	Infraestrutura	<i>infrastructure/repositories/disciplina_repository_impl.py</i>	Converte a entidade de domínio em um “ <i>DisciplinaModel</i> ” do “ <i>SQLAlchemy</i> ” e salva no banco.

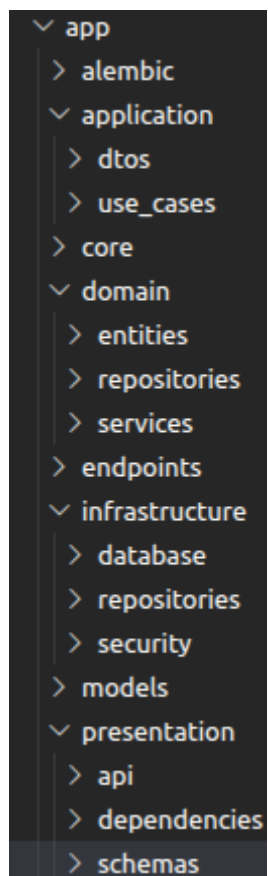
ADAPTANDO A APLICAÇÃO PARA ARQUITETURA LIMPA

Adaptação da Aplicação Criada na Aula Anterior

1) Criando a Estrutura Básica de Pastas

Diretórios Criados

- **application/**
 - dtos/
 - use_cases/
- **domain/**
 - entities/
 - repositories/
 - services/
- **infrastructure/**
 - database/
 - repositories/
 - security/
- **presentation**
 - api/
 - dependencies/
 - schemas/



2) Definindo as Entidades de Domínio (app/domain/entities)

As entidades de domínio são o coração da aplicação, e devem ser classes Python puras, sem qualquer dependência de frameworks web (FastAPI) ou ORMs (SQLAlchemy). Elas representam os objetos de negócio e suas regras mais fundamentais.

Ação: criar as classes de entidade para Curso , Disciplina e User dentro de `app/domain/entities`. Nesta etapa usamos dataclasses ou classes *Pydantic* simples para definir as entidades, tendo como foco apenas os atributos e comportamentos de negócio.

Arquivo: “app/domain/entities/aluno.py”

```
Python
from dataclasses import dataclass, field
from datetime import datetime
from typing import TYPE_CHECKING, Optional

from app.domain.entities.curso import CursoResumo

if TYPE_CHECKING:
    from app.domain.entities.disciplina import DisciplinaResumo

@dataclass(frozen=True)
class AlunoResumo:
    id: int
    nome: str
    turma: int
    curso_id: int | None = None

@dataclass(frozen=True)
class MatriculaDoAluno:
    aluno_id: int
    disciplina_id: int
    disciplina: Optional["DisciplinaResumo"] = None

@dataclass(frozen=True)
class Aluno:
    id: int
    nome: str
    turma: int
    curso_id: int
    criado_em: datetime
    alterado_em: datetime
```



```
curso: CursoResumo  
matricula: list[MatriculaDoAluno] = field(default_factory=list)
```

Arquivo: “app/domain/entities/curso.py”

Python

```
from dataclasses import dataclass, field  
from datetime import datetime  
from typing import TYPE_CHECKING  
  
if TYPE_CHECKING:  
    from app.domain.entities.disciplina import DisciplinaResumo  
  
@dataclass(frozen=True)  
class CursoResumo:  
    id: int  
    nome: str  
    duracao: int  
  
@dataclass(frozen=True)  
class Curso:  
    id: int  
    nome: str  
    duracao: int  
    criado_em: datetime  
    alterado_em: datetime  
    disciplina: list["DisciplinaResumo"] = field(default_factory=list)
```

Arquivo: “app/domain/entities/disciplina.py”

Python

```
from dataclasses import dataclass, field  
from datetime import datetime  
from typing import TYPE_CHECKING, Optional  
  
from app.domain.entities.curso import CursoResumo  
  
if TYPE_CHECKING:  
    from app.domain.entities.aluno import AlunoResumo  
  
@dataclass(frozen=True)
```

```
class DisciplinaResumo:
    id: int
    nome: str
    carga_horaria: int
    curso_id: int | None = None

@dataclass(frozen=True)
class MatriculaDaDisciplina:
    aluno_id: int
    disciplina_id: int
    aluno: Optional["AlunoResumo"] = None

@dataclass(frozen=True)
class Disciplina:
    id: int
    nome: str
    carga_horaria: int
    curso_id: int
    criado_em: datetime
    alterado_em: datetime
    curso: CursoResumo
    matricula: list[MatriculaDaDisciplina] = field(default_factory=list)
```

Arquivo: "app/domain/entities/matricula.py"

Python

```
from dataclasses import dataclass
from datetime import datetime
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from app.domain.entities.aluno import AlunoResumo
    from app.domain.entities.disciplina import DisciplinaResumo

@dataclass(frozen=True)
class Matricula:
    aluno_id: int
    disciplina_id: int
    criado_em: datetime
    aluno: "AlunoResumo"
```

disciplina: "DisciplinaResumo"

Arquivo: "app/domain/entities/user.py"

Python

```
import enum
from dataclasses import dataclass
from uuid import UUID
```

```
class UserRole(str, enum.Enum):
    PROFESSOR = "professor"
    COORDENADOR = "coordenador"
```

```
@dataclass(frozen=True)
```

```
class UserIdentity:
    id: UUID
    email: str
    role: UserRole
```

Arquivo: "app/domain/entities/__init__.py"

Python

```
from app.domain.entities.aluno import Aluno, AlunoResumo,
MatriculaDoAluno
from app.domain.entities.curso import Curso, CursoResumo
from app.domain.entities.disciplina import (
    Disciplina,
    DisciplinaResumo,
    MatriculaDaDisciplina,
)
from app.domain.entities.matricula import Matricula
from app.domain.entities.user import UserIdentity, UserRole

__all__ = [
    "Aluno", "AlunoResumo", "Curso", "CursoResumo", "Disciplina",
    "DisciplinaResumo", "Matricula", "MatriculaDaDisciplina",
    "MatriculaDoAluno", "UserIdentity", "UserRole",
]
```

3) Criando as Interfaces de Repositório (app/domain/repositories)

As interfaces de repositório definem os contratos para a persistência de dados. Elas são classes abstratas que especificam quais operações de CRUD (Create, Read, Update, Delete) podem ser realizadas nas entidades de domínio. A camada de domínio não se importa com a implementação, apenas com o contrato.

Ação: criar as interfaces abstratas para “*AlunoRepository*”, “*CursoRepository*” e “*DisciplinaRepository*” e “*MatriculaRepository*” usando o módulo *abc* (*Abstract Base Classes* - serve para definir Classes Abstratas) do Python.

Arquivo: “app/domain/repositories/aluno_repository.py”

```
Python
from abc import ABC, abstractmethod

from app.domain.entities.aluno import Aluno

class AlunoRepository(ABC):
    @abstractmethod
    async def create_aluno(self, nome: str, turma: int, curso_id: int)
    -> Aluno: ...

    @abstractmethod
    async def list_alunos(self) -> list[Aluno]: ...

    @abstractmethod
    async def get_aluno(self, aluno_id: int) -> Aluno | None: ...

    @abstractmethod
    async def update_aluno(self, aluno_id: int, nome: str, turma: int,
    curso_id: int) -> Aluno | None: ...

    @abstractmethod
    async def delete_aluno(self, aluno_id: int) -> bool: ...
```

Arquivo: “app/domain/repositories/curso_repository.py”

```
Python
from abc import ABC, abstractmethod

from app.domain.entities.curso import Curso

class CursoRepository(ABC):
```

```
@abstractmethod
async def create_curso(self, nome: str, duracao: int) -> Curso: ...

@abstractmethod
async def list_cursos(self) -> list[Curso]: ...

@abstractmethod
async def get_curso(self, curso_id: int) -> Curso | None: ...

@abstractmethod
async def update_curso(self, curso_id: int, nome: str, duracao:
int) -> Curso | None: ...

@abstractmethod
async def delete_curso(self, curso_id: int) -> bool: ...
```

Arquivo: "app/domain/repositories/disciplina_repository.py"

Python

```
from abc import ABC, abstractmethod

from app.domain.entities.disciplina import Disciplina

class DisciplinaRepository(ABC):
    @abstractmethod
    async def create_disciplina(self, nome: str, carga_horaria: int,
curso_id: int) -> Disciplina: ...

    @abstractmethod
    async def list_disciplinas(self) -> list[Disciplina]: ...

    @abstractmethod
    async def get_disciplina(self, disciplina_id: int) -> Disciplina |
None: ...

    @abstractmethod
    async def update_disciplina(
        self, disciplina_id: int, nome: str, carga_horaria: int,
curso_id: int
    ) -> Disciplina | None: ...
```



```
@abstractmethod
async def delete_disciplina(self, disciplina_id: int) -> bool: ...
```

Arquivo: "app/domain/repositories/matricula_repository.py"

Python

```
from abc import ABC, abstractmethod

from app.domain.entities.matricula import Matricula

class MatriculaRepository(ABC):
    @abstractmethod
    async def create_matricula(self, aluno_id: int, disciplina_id: int)
    -> Matricula: ...

    @abstractmethod
    async def list_matriculas(self) -> list[Matricula]: ...

    @abstractmethod
    async def get_matricula(self, aluno_id: int, disciplina_id: int) ->
    Matricula | None: ...

    @abstractmethod
    async def delete_matricula(self, aluno_id: int, disciplina_id: int)
    -> bool: ...
```

Arquivo: "app/domain/repositories/__init__.py"

Python

```
from app.domain.repositories.aluno_repository import AlunoRepository
from app.domain.repositories.curso_repository import CursoRepository
from app.domain.repositories.disciplina_repository import
DisciplinaRepository
from app.domain.repositories.matricula_repository import
MatriculaRepository

__all__ = ["AlunoRepository", "CursoRepository",
"DisciplinaRepository", "MatriculaRepository"]
```

4) Definindo os Serviços (app/domain/services)

O serviço (arquitetura limpa) costuma encapsular uma regra ou operação que não pertence naturalmente a uma única entidade, por exemplo, uma política que combina dados de várias entidades. Não se cria um serviço para cada recurso automaticamente.

O AuthService foi criado para centralizar a autorização por papel - **“ensure_role”** verifica se o papel do usuário está entre os permitidos para aquela ação. O **“RoleChecker”**, na camada de apresentação, obtém o usuário autenticado e chama essa regra. Se o papel não tiver autorização, a aplicação gera um erro Forbidden, convertido em HTTP 403.

Importante: o serviço em questão trata de autorização (o que este usuário pode fazer), não de autenticação (quem é este usuário?). Login, JWT e gerenciamento do usuário ficam nos adaptadores de segurança em **“infrastructure/security”**.

Arquivo: “app/domain/services/auth_service.py”

Python

```
from collections.abc import Iterable

from app.core.exceptions import Forbidden
from app.domain.entities.user import UserRole

class AuthService:
    """Política de autorização por papel, sem dependência de HTTP ou ORM."""

    @staticmethod
    def ensure_role(user_role: str | UserRole, allowed_roles:
Iterable[UserRole]) -> None:
        if user_role not in allowed_roles:
            raise Forbidden()
```

Arquivo: “app/domain/services/__init__.py”

Python

```
from app.domain.services.auth_service import AuthService

__all__ = ["AuthService"]
```

5) Definindo os DTOs da Aplicação (app/application/dtos)

Os DTOs (*Data Transfer Objects*) são objetos simples que carregam dados entre as camadas da aplicação. Eles são usados para entrada e saída dos Casos de Uso, garantindo que a camada de aplicação não dependa diretamente dos schemas da API ou dos modelos do ORM.

Ação: criar os DTOs para Aluno, Curso, Disciplina e Matrícula que representam os dados necessários para as operações de negócio.

Arquivo: "app/application/dtos/aluno_dtos.py"

```
Python
from dataclasses import dataclass

@dataclass(frozen=True)
class AlunoDTO:
    nome: str
    turma: int
    curso_id: int
```

Arquivo: "app/application/dtos/curso_dtos.py"

```
Python
from dataclasses import dataclass

@dataclass(frozen=True)
class CursoDTO:
    nome: str
    duracao: int
```

Arquivo: "app/application/dtos/disciplina_dtos.py"

```
Python
from dataclasses import dataclass

@dataclass(frozen=True)
class DisciplinaDTO:
    nome: str
    carga_horaria: int
    curso_id: int
```

Arquivo: "app/application/dtos/matricula_dtos.py"

Python

```
from dataclasses import dataclass
```

```
@dataclass(frozen=True)
class MatriculaDTO:
    aluno_id: int
    disciplina_id: int
```

Arquivo: "app/application/dtos/__init__.py"

Python

```
from app.application.dtos.aluno import AlunoDTO
from app.application.dtos.curso import CursoDTO
from app.application.dtos.disciplina import DisciplinaDTO
from app.application.dtos.matricula import MatriculaDTO

__all__ = ["AlunoDTO", "CursoDTO", "DisciplinaDTO", "MatriculaDTO"]
```

6) Implementando os Casos de Uso (app/application/use_cases)

Os Casos de Uso contêm a lógica de negócio específica da aplicação. Eles orquestram as operações, utilizando as entidades de domínio e as interfaces de repositório.

Ação: criar as classes de Casos de Uso para as operações de CRUD de Aluno, Curso, Disciplina e Matrícula.

Arquivo: "app/application/use_cases/aluno_use_cases.py"

Python

```
from app.application.dtos.aluno import AlunoDTO
from app.core.exceptions import InvalidReference, NotFound
from app.domain.entities.aluno import Aluno
from app.domain.repositories.aluno_repository import AlunoRepository
from app.domain.repositories.curso_repository import CursoRepository

class AlunoUseCases:
    def __init__(self, repository: AlunoRepository, curso_repository:
CursoRepository):
        self.repository = repository
        self.cursos = curso_repository
```



```
async def criar_aluno(self, dados: AlunoDTO) -> Aluno:
    if await self.cursos.get_curso(dados.curso_id) is None:
        raise InvalidReference(
            f"Não é possível cadastrar o aluno. O curso com ID
{dados.curso_id} não existe."
        )
    return await self.repository.create_aluno(dados.nome,
dados.turma, dados.curso_id)

async def listar_alunos(self) -> list[Aluno]:
    return await self.repository.list_alunos()

async def buscar_aluno(self, aluno_id: int) -> Aluno:
    aluno = await self.repository.get_aluno(aluno_id)
    if aluno is None:
        raise NotFound("Aluno não encontrado")
    return aluno

async def atualizar_aluno(self, aluno_id: int, dados: AlunoDTO) ->
Aluno:
    if await self.repository.get_aluno(aluno_id) is None:
        raise NotFound("Aluno não encontrado")
    if await self.cursos.get_curso(dados.curso_id) is None:
        raise InvalidReference(f"Curso com ID {dados.curso_id} não
existe.")
    aluno = await self.repository.update_aluno(
        aluno_id, dados.nome, dados.turma, dados.curso_id
    )
    if aluno is None:
        raise NotFound("Aluno não encontrado")
    return aluno

async def deletar_aluno(self, aluno_id: int) -> None:
    if not await self.repository.delete_aluno(aluno_id):
        raise NotFound("Aluno não encontrado")
```

Arquivo: "app/application/use_cases/curso_use_cases.py"

Python

```
from app.application.dtos.curso import CursoDTO
from app.core.exceptions import NotFound
```



```
from app.domain.repositories.curso_repository import CursoRepository
from app.domain.entities.curso import Curso

class CursoUseCases:
    def __init__(self, repository: CursoRepository):
        self.repository = repository

    async def criar_curso(self, dados: CursoDTO) -> Curso:
        return await self.repository.create_curso(dados.nome,
        dados.duracao)

    async def listar_cursos(self) -> list[Curso]:
        return await self.repository.list_cursos()

    async def buscar_curso(self, curso_id: int) -> Curso:
        curso = await self.repository.get_curso(curso_id)
        if curso is None:
            raise NotFound("Curso não encontrado")
        return curso

    async def atualizar_curso(self, curso_id: int, dados: CursoDTO) ->
Curso:
        curso = await self.repository.update_curso(curso_id,
        dados.nome, dados.duracao)
        if curso is None:
            raise NotFound("Curso não encontrado")
        return curso

    async def deletar_curso(self, curso_id: int) -> None:
        if not await self.repository.delete_curso(curso_id):
            raise NotFound("Curso não encontrado")
```

Arquivo: "app/application/use_cases/disciplina_use_cases.py"

Python

```
from app.application.dtos.disciplina import DisciplinaDTO
from app.core.exceptions import InvalidReference, NotFound
from app.domain.entities.disciplina import Disciplina
from app.domain.repositories.curso_repository import CursoRepository
```



```
from app.domain.repositories.disciplina_repository import
DisciplinaRepository

class DisciplinaUseCases:
    def __init__(self, repository: DisciplinaRepository,
curso_repository: CursoRepository):
        self.repository = repository
        self.cursos = curso_repository

    async def criar_disciplina(self, dados: DisciplinaDTO) ->
Disciplina:
        if await self.cursos.get_curso(dados.curso_id) is None:
            raise InvalidReference(
                f"Não é possível criar a disciplina. O curso com ID
{dados.curso_id} não existe."
            )
        return await self.repository.create_disciplina(
            dados.nome, dados.carga_horaria, dados.curso_id
        )

    async def listar_disciplinas(self) -> list[Disciplina]:
        return await self.repository.list_disciplinas()

    async def buscar_disciplina(self, disciplina_id: int) ->
Disciplina:
        disciplina = await
self.repository.get_disciplina(disciplina_id)
        if disciplina is None:
            raise NotFound("Disciplina não encontrada")
        return disciplina

    async def atualizar_disciplina(self, disciplina_id: int, dados:
DisciplinaDTO) -> Disciplina:
        if await self.repository.get_disciplina(disciplina_id) is None:
            raise NotFound("Disciplina não encontrada")
        if await self.cursos.get_curso(dados.curso_id) is None:
            raise InvalidReference(f"Curso com ID {dados.curso_id} não
existe.")
        disciplina = await self.repository.update_disciplina(
            disciplina_id, dados.nome, dados.carga_horaria,
dados.curso_id
```



```
)  
if disciplina is None:  
    raise NotFound("Disciplina não encontrada")  
return disciplina  
  
async def deletar_disciplina(self, disciplina_id: int) -> None:  
    if not await self.repository.delete_disciplina(disciplina_id):  
        raise NotFound("Disciplina não encontrada")
```

Arquivo: "app/application/use_cases/matricula_use_cases.py"

Python

```
from app.application.dtos.matricula import MatriculaDTO  
from app.core.exceptions import InvalidReference, NotFound  
from app.domain.entities.matricula import Matricula  
from app.domain.repositories.aluno_repository import AlunoRepository  
from app.domain.repositories.disciplina_repository import  
DisciplinaRepository  
from app.domain.repositories.matricula_repository import  
MatriculaRepository  
  
class MatriculaUseCases:  
    def __init__(  
        self,  
        repository: MatriculaRepository,  
        aluno_repository: AlunoRepository,  
        disciplina_repository: DisciplinaRepository,  
    ):  
        self.repository = repository  
        self.alunos = aluno_repository  
        self.disciplinas = disciplina_repository  
  
    async def criar_matricula(self, dados: MatriculaDTO) -> Matricula:  
        if await self.alunos.get_aluno(dados.aluno_id) is None:  
            raise InvalidReference(  
                f"Não é possível efetuar a matrícula. O aluno com ID  
{dados.aluno_id} não existe."  
            )  
        if await self.disciplinas.get_disciplina(dados.disciplina_id)  
is None:
```



```
        raise InvalidReference(
            f"Não é possível efetuar a matrícula. A disciplina com
            ID {dados.disciplina_id} não existe."
        )
    return await self.repository.create_matricula(dados.aluno_id,
        dados.disciplina_id)

    async def listar_matriculas(self) -> list[Matricula]:
        return await self.repository.list_matriculas()

    async def buscar_matricula(self, aluno_id: int, disciplina_id: int)
    -> Matricula:
        matricula = await self.repository.get_matricula(aluno_id,
            disciplina_id)
        if matricula is None:
            raise NotFound("Matrícula não encontrada!")
        return matricula

    async def deletar_matricula(self, aluno_id: int, disciplina_id:
    int) -> None:
        if not await self.repository.delete_matricula(aluno_id,
            disciplina_id):
            raise NotFound("Matrícula não encontrada")
```

Arquivo: "app/application/use_cases/__init__.py"

Python

```
from app.application.use_cases.aluno import AlunoUseCases
from app.application.use_cases.curso import CursoUseCases
from app.application.use_cases.disciplina import DisciplinaUseCases
from app.application.use_cases.matricula import MatriculaUseCases

__all__ = ["AlunoUseCases", "CursoUseCases", "DisciplinaUseCases",
    "MatriculaUseCases"]
```

7) Configurando a Infraestrutura (app/infrastructure)

Esta camada é responsável por implementar os detalhes técnicos, como a interação com o banco de dados via "SQLAlchemy" e a configuração da segurança com "fastapi-users".

⇒ Modelos do SQLAlchemy (app/infrastructure/database/models.py)

Os modelos do SQLAlchemy são as classes que mapeiam diretamente as tabelas do banco de dados. Eles devem ser separados das entidades de domínio.

Ação: mover os modelos “*Aluno*”, “*Curso*”, “*Disciplina*”, “*Matricula*” e “*User*” existentes de “*app/models*” para “*app/infrastructure/database/models.py*” e renomear para “*AlunoModel*”, “*CursoModel*”, “*DisciplinaModel*”, “*MatriculaModel*” e “*UserModel*”, esse processo evita possíveis conflitos com as entidades de domínio.

Arquivo: “app/infrastructure/database/models.py”

Python

```
import uuid
from datetime import datetime, timezone

from sqlalchemy import Boolean, DateTime, ForeignKey, Integer, String,
UUID
from sqlalchemy.orm import Mapped, mapped_column, relationship

from app.domain.entities.user import UserRole
from app.infrastructure.database.session import Base

class CursoModel(Base):
    __tablename__ = "cursos"

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
index=True, autoincrement=True)
    nome: Mapped[str] = mapped_column(String, nullable=False)
    duracao: Mapped[int] = mapped_column(Integer, nullable=False)
    criado_em: Mapped[datetime] = mapped_column(DateTime,
default=lambda: datetime.now(timezone.utc))
    alterado_em: Mapped[datetime] = mapped_column(
        DateTime, default=lambda: datetime.now(timezone.utc),
        onupdate=lambda: datetime.now(timezone.utc),
    )
    aluno: Mapped[list["AlunoModel"]] = relationship(
        back_populates="curso", cascade="all, delete-orphan"
    )
    disciplina: Mapped[list["DisciplinaModel"]] = relationship(
        back_populates="curso", cascade="all, delete-orphan"
    )

class AlunoModel(Base):
```

```
__tablename__ = "alunos"

id: Mapped[int] = mapped_column(Integer, primary_key=True,
index=True, autoincrement=True)
nome: Mapped[str] = mapped_column(String, nullable=False)
turma: Mapped[int] = mapped_column(Integer, nullable=False)
curso_id: Mapped[int] = mapped_column(
    Integer, ForeignKey("cursos.id", ondelete="CASCADE"),
    nullable=False
)
criado_em: Mapped[datetime] = mapped_column(DateTime,
default=lambda: datetime.now(timezone.utc))
alterado_em: Mapped[datetime] = mapped_column(
    DateTime, default=lambda: datetime.now(timezone.utc),
    onupdate=lambda: datetime.now(timezone.utc),
)
curso: Mapped[CursoModel] = relationship(back_populates="aluno")
matricula: Mapped[list["MatriculaModel"]] =
relationship(back_populates="aluno")

class DisciplinaModel(Base):
    __tablename__ = "disciplinas"

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
index=True, autoincrement=True)
    nome: Mapped[str] = mapped_column(String, nullable=False)
    carga_horaria: Mapped[int] = mapped_column(Integer, nullable=False)
    curso_id: Mapped[int] = mapped_column(
        Integer, ForeignKey("cursos.id", ondelete="CASCADE"),
        nullable=False
    )
    criado_em: Mapped[datetime] = mapped_column(DateTime,
default=lambda: datetime.now(timezone.utc))
    alterado_em: Mapped[datetime] = mapped_column(
        DateTime, default=lambda: datetime.now(timezone.utc),
        onupdate=lambda: datetime.now(timezone.utc),
    )
    curso: Mapped[CursoModel] =
relationship(back_populates="disciplina")
    matricula: Mapped[list["MatriculaModel"]] =
relationship(back_populates="disciplina")
```

```
class MatriculaModel(Base):
    __tablename__ = "matriculas"

    aluno_id: Mapped[int] = mapped_column(
        Integer, ForeignKey("alunos.id", ondelete="CASCADE"),
        primary_key=True
    )
    disciplina_id: Mapped[int] = mapped_column(
        Integer, ForeignKey("disciplinas.id", ondelete="CASCADE"),
        primary_key=True
    )
    criado_em: Mapped[datetime] = mapped_column(DateTime,
        default=lambda: datetime.now(timezone.utc))
    aluno: Mapped[AlunoModel] =
        relationship(back_populates="matricula")
    disciplina: Mapped[DisciplinaModel] =
        relationship(back_populates="matricula")

class UserModel(Base):
    __tablename__ = "users"

    id: Mapped[uuid.UUID] = mapped_column(UUID(as_uuid=True),
        primary_key=True, default=uuid.uuid4)
    email: Mapped[str] = mapped_column(String(length=320), unique=True,
        index=True, nullable=False)
    hashed_password: Mapped[str] = mapped_column(String(length=1024),
        nullable=False)
    is_active: Mapped[bool] = mapped_column(Boolean, default=True,
        nullable=False)
    is_superuser: Mapped[bool] = mapped_column(Boolean, default=False,
        nullable=False)
    is_verified: Mapped[bool] = mapped_column(Boolean, default=False,
        nullable=False)
    role: Mapped[UserRole] = mapped_column(String(20), nullable=False,
        default=UserRole.PROFESSOR)
```

⇒ Sessão do Banco de Dados (app/infrastructure/database/session.py)

O arquivo database.py existente será renomeado e ajustado para a nova estrutura.

Ação: Mover o conteúdo de “*app/database.py*” para “*app/infrastructure/database/session.py*” e ajustar os imports.

Arquivo: “app/infrastructure/database/session.py”

```
Python
from collections.abc import AsyncGenerator

from sqlalchemy.ext.asyncio import AsyncSession, async_sessionmaker,
create_async_engine
from sqlalchemy.orm import DeclarativeBase

from app.core.config import DATABASE_URL

engine = create_async_engine(DATABASE_URL)
async_session_maker = async_sessionmaker(engine,
expire_on_commit=False)

class Base(DeclarativeBase):
    """Base declarativa dos modelos SQLAlchemy."""

    __allow_unmapped__ = True

    async def get_db() -> AsyncGenerator[AsyncSession, None]:
        async with async_session_maker() as session:
            yield session

get_async_session = get_db
```

⇒ Implementando Repositório (app/infrastructure/repositories)

Neste momento, as interfaces de repositório definidas na camada de domínio serão implementadas usando o *SQLAlchemy*. Esta é a camada que traduz as operações de domínio para operações de banco de dados.

Ação: Criar as implementações “*curso_repository.py*”, “*curso_repository.py*”, “*disciplina_repository.py*” e “*matricula_repository.py*”.

Ação: criar os mapeadores (arquivo *_mappers.py*), que contêm o código responsável por converter dados entre os Modelos do Banco de Dados (ORM) e as Entidades de Domínio.

O papel do Mapper na arquitetura

- Entidade de Domínio (Domínio puro): Representa as regras de negócio. É uma classe Python pura (ou modelo Pydantic) sem qualquer vínculo com tabelas, SQL ou bibliotecas como SQLAlchemy.
- Modelo ORM (Infraestrutura): Representa a estrutura da tabela no banco de dados (ex: classes que herdam de DeclarativeBase do SQLAlchemy).
- Mapper (Tradutor): Faz o de-para entre essas duas representações dentro da camada de repositório.

Exemplo de fluxo

- Na leitura (Banco $\Rightarrow$ Domínio): o repositório busca o registro usando o SQLAlchemy (retorna UserModel), o mapper converte esse registro em uma entidade de domínio (User), e o repositório devolve a entidade para o Caso de Uso (Use Case).
- Na escrita (Domínio $\Rightarrow$ Banco): o Caso de Uso envia a entidade (User) para o repositório. O mapper converte essa entidade em um modelo de tabela (UserModel), que o repositório insere no banco.

Arquivo: “app/infrastructure/repositories/aluno_repository_impl.py”

Python

```
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import selectinload

from app.domain.entities.aluno import Aluno
from app.domain.repositories.aluno_repository import AlunoRepository
from app.infrastructure.database.models import AlunoModel,
MatriculaModel
from app.infrastructure.repositories._mappers import map_aluno

class AlunoRepositoryImpl(AlunoRepository):
    def __init__(self, session: AsyncSession):
        self.session = session
```



```
@staticmethod
def _statement():
    return select(AlunoModel).options(
        selectinload(AlunoModel.curso),

selectinload(AlunoModel.matricula).selectinload(MatriculaModel.disciplina),
    )

    async def create_aluno(self, nome: str, turma: int, curso_id: int)
-> Aluno:
    row = AlunoModel(nome=nome, turma=turma, curso_id=curso_id)
    self.session.add(row)
    await self.session.commit()
    result = await self.get_aluno(row.id)
    assert result is not None
    return result

    async def list_alunos(self) -> list[Aluno]:
    rows = (await self.session.scalars(self._statement())).all()
    return [map_aluno(row) for row in rows]

    async def get_aluno(self, aluno_id: int) -> Aluno | None:
    row = await
self.session.scalar(self._statement().where(AlunoModel.id == aluno_id))
    return map_aluno(row) if row else None

    async def update_aluno(
        self, aluno_id: int, nome: str, turma: int, curso_id: int
    ) -> Aluno | None:
    row = await self.session.get(AlunoModel, aluno_id)
    if row is None:
        return None
    row.nome, row.turma, row.curso_id = nome, turma, curso_id
    await self.session.commit()
    return await self.get_aluno(aluno_id)

    async def delete_aluno(self, aluno_id: int) -> bool:
    row = await self.session.get(AlunoModel, aluno_id)
    if row is None:
        return False
    await self.session.delete(row)
```



```
await self.session.commit()
return True
```

Arquivo: “app/infrastructure/repositories/curso_repository_impl.py”

Python

```
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import selectinload

from app.domain.entities.curso import Curso
from app.domain.repositories.curso_repository import CursoRepository
from app.infrastructure.database.models import CursoModel
from app.infrastructure.repositories._mappers import map_curso

class CursoRepositoryImpl(CursoRepository):
    def __init__(self, session: AsyncSession):
        self.session = session

    @staticmethod
    def _statement():
        return
select(CursoModel).options(selectinload(CursoModel.disciplina))

    async def create_curso(self, nome: str, duracao: int) -> Curso:
        row = CursoModel(nome=nome, duracao=duracao)
        self.session.add(row)
        await self.session.commit()
        result = await self.get_curso(row.id)
        assert result is not None
        return result

    async def list_cursos(self) -> list[Curso]:
        rows = (await self.session.scalars(self._statement())).all()
        return [map_curso(row) for row in rows]

    async def get_curso(self, curso_id: int) -> Curso | None:
        row = await
self.session.scalar(self._statement().where(CursoModel.id == curso_id))
        return map_curso(row) if row else None
```



```
    async def update_curso(self, curso_id: int, nome: str, duracao:
int) -> Curso | None:
        row = await self.session.get(CursoModel, curso_id)
        if row is None:
            return None
        row.nome, row.duracao = nome, duracao
        await self.session.commit()
        return await self.get_curso(curso_id)

    async def delete_curso(self, curso_id: int) -> bool:
        row = await self.session.get(CursoModel, curso_id)
        if row is None:
            return False
        await self.session.delete(row)
        await self.session.commit()
        return True
```

Arquivo: “app/infrastructure/repositories/disciplina_repository_impl.py”

Python

```
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import selectinload

from app.domain.entities.disciplina import Disciplina
from app.domain.repositories.disciplina_repository import
DisciplinaRepository
from app.infrastructure.database.models import DisciplinaModel,
MatriculaModel
from app.infrastructure.repositories._mappers import map_disciplina

class DisciplinaRepositoryImpl(DisciplinaRepository):
    def __init__(self, session: AsyncSession):
        self.session = session

    @staticmethod
    def _statement():
        return select(DisciplinaModel).options(
            selectinload(DisciplinaModel.curso),
```



```
selectinload(DisciplinaModel.matricula).selectinload(MatriculaModel.alu  
no),  
    )
```

```
    async def create_disciplina(  
        self, nome: str, carga_horaria: int, curso_id: int  
    ) -> Disciplina:  
        row = DisciplinaModel(nome=nome, carga_horaria=carga_horaria,  
curso_id=curso_id)  
        self.session.add(row)  
        await self.session.commit()  
        result = await self.get_disciplina(row.id)  
        assert result is not None  
        return result
```

```
    async def list_disciplinas(self) -> list[Disciplina]:  
        rows = (await self.session.scalars(self._statement())).all()  
        return [map_disciplina(row) for row in rows]
```

```
    async def get_disciplina(self, disciplina_id: int) -> Disciplina |  
None:  
        row = await self.session.scalar(  
            self._statement().where(DisciplinaModel.id ==  
disciplina_id)  
        )  
        return map_disciplina(row) if row else None
```

```
    async def update_disciplina(  
        self, disciplina_id: int, nome: str, carga_horaria: int,  
curso_id: int  
    ) -> Disciplina | None:  
        row = await self.session.get(DisciplinaModel, disciplina_id)  
        if row is None:  
            return None  
        row.nome, row.carga_horaria, row.curso_id = nome,  
carga_horaria, curso_id  
        await self.session.commit()  
        return await self.get_disciplina(disciplina_id)
```

```
    async def delete_disciplina(self, disciplina_id: int) -> bool:  
        row = await self.session.get(DisciplinaModel, disciplina_id)
```



```
if row is None:
    return False
await self.session.delete(row)
await self.session.commit()
return True
```

Arquivo: "app/infrastructure/repositories/matricula_repository_impl.py"

Python

```
from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import selectinload

from app.domain.entities.matricula import Matricula
from app.domain.repositories.matricula_repository import
MatriculaRepository
from app.infrastructure.database.models import AlunoModel,
DisciplinaModel, MatriculaModel
from app.infrastructure.repositories._mappers import map_matricula

class MatriculaRepositoryImpl(MatriculaRepository):
    def __init__(self, session: AsyncSession):
        self.session = session

    @staticmethod
    def _statement():
        return select(MatriculaModel).options(
            selectinload(MatriculaModel.aluno),
            selectinload(MatriculaModel.disciplina),
        )

    async def create_matricula(self, aluno_id: int, disciplina_id: int)
-> Matricula:
        row = MatriculaModel(aluno_id=aluno_id,
disciplina_id=disciplina_id)
        self.session.add(row)
        await self.session.commit()
        result = await self.get_matricula(aluno_id, disciplina_id)
        assert result is not None
        return result
```



```
async def list_matriculas(self) -> list[Matricula]:
    rows = (await self.session.scalars(self._statement())).all()
    return [map_matricula(row) for row in rows]

async def get_matricula(self, aluno_id: int, disciplina_id: int) ->
Matricula | None:
    row = await self.session.scalar(
        self._statement().where(
            MatriculaModel.aluno_id == aluno_id,
            MatriculaModel.disciplina_id == disciplina_id,
        )
    )
    return map_matricula(row) if row else None

async def delete_matricula(self, aluno_id: int, disciplina_id: int)
-> bool:
    row = await self.session.get(MatriculaModel, (aluno_id,
disciplina_id))
    if row is None:
        return False
    await self.session.delete(row)
    await self.session.commit()
    return True
```

Arquivo: "app/infrastructure/repositories/_mappers.py"

Python

```
from app.domain.entities.aluno import Aluno, AlunoResumo,
MatriculaDoAluno
from app.domain.entities.curso import Curso, CursoResumo
from app.domain.entities.disciplina import (
    Disciplina, DisciplinaResumo, MatriculaDaDisciplina,
)
from app.domain.entities.matricula import Matricula
from app.infrastructure.database.models import (
    AlunoModel, CursoModel, DisciplinaModel, MatriculaModel,
)

def curso_resumo(row: CursoModel) -> CursoResumo:
    return CursoResumo(row.id, row.nome, row.duracao)
```



```
def disciplina_resumo(row: DisciplinaModel) -> DisciplinaResumo:  
    return DisciplinaResumo(row.id, row.nome, row.carga_horaria,  
row.curso_id)
```

```
def aluno_resumo(row: AlunoModel) -> AlunoResumo:  
    return AlunoResumo(row.id, row.nome, row.turma, row.curso_id)
```

```
def map_curso(row: CursoModel) -> Curso:  
    return Curso(  
        id=row.id,  
        nome=row.nome,  
        duracao=row.duracao,  
        criado_em=row.criado_em,  
        alterado_em=row.alterado_em,  
        disciplina=[disciplina_resumo(item) for item in  
row.disciplina],  
    )
```

```
def map_aluno(row: AlunoModel) -> Aluno:  
    return Aluno(  
        id=row.id,  
        nome=row.nome,  
        turma=row.turma,  
        curso_id=row.curso_id,  
        criado_em=row.criado_em,  
        alterado_em=row.alterado_em,  
        curso=curso_resumo(row.curso),  
        matricula=[  
            MatriculaDoAluno(  
                aluno_id=item.aluno_id,  
                disciplina_id=item.disciplina_id,  
                disciplina=disciplina_resumo(item.disciplina),  
            )  
            for item in row.matricula  
        ],  
    )
```

```
def map_disciplina(row: DisciplinaModel) -> Disciplina:  
    return Disciplina(  
        id=row.id,  
        nome=row.nome,
```



```
carga_horaria=row.carga_horaria,
curso_id=row.curso_id,
criado_em=row.criado_em,
alterado_em=row.alterado_em,
curso=curso_resumo(row.curso),
matricula=[
    MatriculaDaDisciplina(
        aluno_id=item.aluno_id,
        disciplina_id=item.disciplina_id,
        aluno=aluno_resumo(item.aluno),
    )
    for item in row.matricula
],
)

def map_matricula(row: MatriculaModel) -> Matricula:
    return Matricula(
        aluno_id=row.aluno_id,
        disciplina_id=row.disciplina_id,
        criado_em=row.criado_em,
        aluno=aluno_resumo(row.aluno),
        disciplina=disciplina_resumo(row.disciplina),
    )
```

Arquivo: "app/infrastructure/repositories/__init__.py"

Python

```
from app.infrastructure.repositories.aluno_repository_impl import
AlunoRepositoryImpl
from app.infrastructure.repositories.curso_repository_impl import
CursoRepositoryImpl
from app.infrastructure.repositories.disciplina_repository_impl import
DisciplinaRepositoryImpl
from app.infrastructure.repositories.matricula_repository_impl import
MatriculaRepositoryImpl

__all__ = [
    "AlunoRepositoryImpl", "CursoRepositoryImpl",
    "DisciplinaRepositoryImpl",
    "MatriculaRepositoryImpl",
]
```

⇒ Efetuada a Configuração de Segurança (app/infrastructure/security)

Os arquivos “*security.py*”, “*user_manager.py*” e “*password_hash.py*” da pasta “*app/core*” serão movidos e ajustados para a “*camada de infraestrutura*”, pois contêm detalhes de implementação do “*fastapi-users*”.

Ação: mover e ajustar os arquivos para “*app/infrastructure/security/*”

IMPORTANTE: A lógica foi movida porque se decidirmos, futuramente, parar de usar JWT e passarmos a usar Cookies, ou se trocarmos o “*fastapi-users*” por outra biblioteca, não será necessário mexer no ‘coração’ da aplicação. Ao mover a segurança para “*infrastructure/security*”, deixamos claro que a segurança é um serviço que a infraestrutura provê para a aplicação, e não uma regra de negócio central.”

Arquivo: “app/infrastructure/security/auth_backend.py”

```
Python
from fastapi_users.authentication import AuthenticationBackend,
BearerTransport, JWTStrategy

from app.core.config import JWT_LIFETIME_SECONDS, JWT_SECRET

bearer_transport = BearerTransport(tokenUrl="auth/jwt/login")

def get_jwt_strategy() -> JWTStrategy:
    return JWTStrategy(secret=JWT_SECRET,
lifetime_seconds=JWT_LIFETIME_SECONDS)

auth_backend = AuthenticationBackend(
    name="jwt",
    transport=bearer_transport,
    get_strategy=get_jwt_strategy,
)
```

Arquivo: “app/infrastructure/security/user_manager.py”

```
Python
from typing import Optional
from uuid import UUID

from fastapi import Depends, Request
```



```
from fastapi_users import BaseUserManager, FastAPIUsers, UUIDIDMixin
from fastapi_users_db_sqlalchemy import SQLAlchemyUserDatabase
from sqlalchemy.ext.asyncio import AsyncSession

from app.core.config import RESET_PASSWORD_TOKEN_SECRET,
VERIFICATION_TOKEN_SECRET
from app.infrastructure.database.session import get_db
from app.infrastructure.database.models import UserModel
from app.infrastructure.security.auth_backend import auth_backend
from app.infrastructure.security.password_hasher import password_helper

async def get_user_db(session: AsyncSession = Depends(get_db)):
    yield SQLAlchemyUserDatabase(session, UserModel)

class UserManager(UUIDIDMixin, BaseUserManager[UserModel, UUID]):
    reset_password_token_secret = RESET_PASSWORD_TOKEN_SECRET
    verification_token_secret = VERIFICATION_TOKEN_SECRET
    password_helper = password_helper

    async def on_after_register(self, user: UserModel, request:
Optional[Request] = None):
        print(f"Usuário {user.id} foi registrado com sucesso.")

def get_user_manager(user_db=Depends(get_user_db)):
    yield UserManager(user_db)

fastapi_users = FastAPIUsers[UserModel, UUID](get_user_manager,
[auth_backend])
current_active_user = fastapi_users.current_user(active=True)
```

Arquivo: "app/models/password_hasher.py"

Python

```
from fastapi_users.password import PasswordHelper

password_helper = PasswordHelper()

class PasswordHasher:
    """Adaptador simples da estratégia de hashing padrão do FastAPI
    Users."""
```



```
def hash(self, password: str) -> str:
    return password_helper.hash(password)

def verify_and_update(self, plain_password: str, hashed_password:
str):
    return password_helper.verify_and_update(plain_password,
hashed_password)
```

Arquivo: "app/models/__init__.py"

Python

```
from app.infrastructure.security.auth_backend import auth_backend
from app.infrastructure.security.user_manager import
current_active_user, fastapi_users

__all__ = ["auth_backend", "current_active_user", "fastapi_users"]
```

8) Refatorando a Camada de Apresentação (app/presentation)

A camada de apresentação será responsável por receber as requisições HTTP, validar os dados, invocar os Casos de Uso e retornar as respostas. Ela deve ser o mais leve possível, com pouca lógica de negócio.

⇒ Schemas Pydantic para API (app/presentation/schemas)

Estes schemas são para a interface externa da API e podem ser diferentes dos DTOs da camada de aplicação.

Ação: criar "*schemas Pydantic*" para Aluno, Curso, Disciplina, Matrícula e User.

Arquivo: "app/presentation/schemas/aluno_schemas.py"

Python

```
from datetime import datetime
from typing import Optional

from pydantic import BaseModel, ConfigDict

from app.presentation.schemas.curso_schemas import CursoResumoSchema
```



```
class DisciplinaDoAlunoSchema(BaseModel):
    id: int
    nome: str
    model_config = ConfigDict(from_attributes=True)

class MatriculaDoAlunoSchema(BaseModel):
    aluno_id: int
    disciplina_id: int
    disciplina: Optional[DisciplinaDoAlunoSchema] = None
    model_config = ConfigDict(from_attributes=True)

class AlunoCreate(BaseModel):
    nome: str
    turma: int
    curso_id: int

class AlunoResponse(BaseModel):
    id: int
    nome: str
    turma: int
    curso_id: int
    criado_em: datetime
    alterado_em: datetime
    curso: CursoResumoSchema
    matricula: list[MatriculaDoAlunoSchema] = []
    model_config = ConfigDict(from_attributes=True)
```

Arquivo: “app/presentation/schemas/curso_schemas.py”

Python

```
from datetime import datetime

from pydantic import BaseModel, ConfigDict

class DisciplinaResumoSchema(BaseModel):
    id: int
    nome: str
```

```
carga_horaria: int
model_config = ConfigDict(from_attributes=True)

class CursoResumoSchema(BaseModel):
    id: int
    nome: str
    duracao: int
    model_config = ConfigDict(from_attributes=True)

class CursoCreate(BaseModel):
    nome: str
    duracao: int

class CursoResponse(BaseModel):
    id: int
    nome: str
    duracao: int
    criado_em: datetime
    alterado_em: datetime
    disciplina: list[DisciplinaResumoSchema] = []
    model_config = ConfigDict(from_attributes=True)
```

Arquivo: "app/presentation/schemas/disciplina_schemas.py"

```
Python
from datetime import datetime
from typing import Optional

from pydantic import BaseModel, ConfigDict

from app.presentation.schemas.curso_schemas import CursoResumoSchema

class AlunoResumoSchema(BaseModel):
    id: int
    nome: str
    turma: int
    model_config = ConfigDict(from_attributes=True)
```

```
class MatriculaDaDisciplinaSchema(BaseModel):
    aluno_id: int
    disciplina_id: int
    aluno: Optional[AlunoResumoSchema] = None
    model_config = ConfigDict(from_attributes=True)

class DisciplinaCreate(BaseModel):
    nome: str
    carga_horaria: int
    curso_id: int

class DisciplinaResponse(BaseModel):
    id: int
    nome: str
    carga_horaria: int
    curso_id: int
    criado_em: datetime
    alterado_em: datetime
    curso: CursoResumoSchema
    matricula: list[MatriculaDaDisciplinaSchema] = []
    model_config = ConfigDict(from_attributes=True)
```

Arquivo: "app/presentation/schemas/matricula_schemas.py"

Python

```
from datetime import datetime

from pydantic import BaseModel, ConfigDict

class AlunoDaMatriculaSchema(BaseModel):
    id: int
    nome: str
    turma: int
    curso_id: int
    model_config = ConfigDict(from_attributes=True)

class DisciplinaDaMatriculaSchema(BaseModel):
    id: int
    nome: str
    carga_horaria: int
    curso_id: int
```



```
model_config = ConfigDict(from_attributes=True)

class MatriculaCreate(BaseModel):
    aluno_id: int
    disciplina_id: int

class MatriculaResponse(BaseModel):
    aluno_id: int
    disciplina_id: int
    criado_em: datetime
    aluno: AlunoDaMatriculaSchema
    disciplina: DisciplinaDaMatriculaSchema
    model_config = ConfigDict(from_attributes=True)
```

Arquivo: "app/presentation/schemas/user_schemas.py"

Python

```
from typing import Optional
from uuid import UUID

from fastapi_users import schemas
from pydantic import ConfigDict

from app.domain.entities.user import UserRole

class UserRead(schemas.BaseUser[UUID]):
    role: UserRole
    model_config = ConfigDict(from_attributes=True)

class UserCreate(schemas.BaseUserCreate):
    role: UserRole = UserRole.PROFESSOR

class UserUpdate(schemas.BaseUserUpdate):
    role: Optional[UserRole] = None
```

Arquivo: "app/presentation/schemas/__init__.py"

Python

```
from app.presentation.schemas.aluno_schemas import AlunoCreate,
AlunoResponse
```

```
from app.presentation.schemas.curso_schemas import CursoCreate,
CursoResponse, CursoResumoSchema
from app.presentation.schemas.disciplina_schemas import
DisciplinaCreate, DisciplinaResponse
from app.presentation.schemas.matricula_schemas import MatriculaCreate,
MatriculaResponse
from app.presentation.schemas.user_schemas import UserCreate, UserRead,
UserUpdate

__all__ = [
    "AlunoCreate", "AlunoResponse", "CursoCreate", "CursoResponse",
    "CursoResumoSchema", "DisciplinaCreate", "DisciplinaResponse",
    "MatriculaCreate", "MatriculaResponse", "UserCreate", "UserRead",
    "UserUpdate",
]
```

⇒ Efetuando Injeção de Dependência (*app/presentation/dependencies*)

As dependências do *FastAPI* serão usadas para injetar as implementações concretas dos repositórios e os casos de uso nos *endpoints*. Isso é crucial para a Inversão de Controle.

Ação: criar funções de dependência para os repositórios e casos de uso.

Arquivo: *“app/presentation/dependencies/__init__.py”*

```
Python
from fastapi import Depends
from sqlalchemy.ext.asyncio import AsyncSession

from app.application.use_cases import (
    AlunoUseCases, CursoUseCases, DisciplinaUseCases,
    MatriculaUseCases,
)
from app.domain.entities.user import UserRole
from app.domain.services.auth_service import AuthService
from app.infrastructure.database.session import get_db
from app.infrastructure.database.models import UserModel
from app.infrastructure.repositories import (
    AlunoRepositoryImpl, CursoRepositoryImpl, DisciplinaRepositoryImpl,
    MatriculaRepositoryImpl,
)
```



```
from app.infrastructure.security.user_manager import
current_active_user

def get_curso_use_cases(session: AsyncSession = Depends(get_db)) ->
CursoUseCases:
    return CursoUseCases(CursoRepositoryImpl(session))

def get_aluno_use_cases(session: AsyncSession = Depends(get_db)) ->
AlunoUseCases:
    return AlunoUseCases(AlunoRepositoryImpl(session),
CursoRepositoryImpl(session))

def get_disciplina_use_cases(session: AsyncSession = Depends(get_db))
-> DisciplinaUseCases:
    return DisciplinaUseCases(
        DisciplinaRepositoryImpl(session), CursoRepositoryImpl(session)
    )

def get_matricula_use_cases(session: AsyncSession = Depends(get_db)) ->
MatriculaUseCases:
    return MatriculaUseCases(
        MatriculaRepositoryImpl(session),
        AlunoRepositoryImpl(session),
        DisciplinaRepositoryImpl(session),
    )

class RoleChecker:
    def __init__(self, allowed_roles: list[UserRole]):
        self.allowed_roles = allowed_roles

    def __call__(self, user: UserModel = Depends(current_active_user))
-> UserModel:
        AuthService.ensure_role(user.role, self.allowed_roles)
        return user

requer_coordenador = RoleChecker([UserRole.COORDENADOR])
requer_professor = RoleChecker([UserRole.PROFESSOR])
requer_professor_ou_coordenador = RoleChecker(
    [UserRole.PROFESSOR, UserRole.COORDENADOR]
)
```

⇒ Definindo os Routers *FastAPI* (*app/presentation/api*)

Os routers do *FastAPI* são os pontos de entrada para as requisições HTTP. Eles usarão as dependências para obter os casos de uso e executar a lógica de negócio.

Ação: refatorar os arquivos *aluno.py*, *curso.py*, *disciplina.py*, *matricula.py* e *auth.py* para a nova estrutura, utilizando os casos de uso e schemas definidos.

Arquivo: “*app/presentation/api/aluno_router.py*”

Python

```
from fastapi import APIRouter, Depends, status

from app.application.dtos.aluno import AlunoDTO
from app.application.use_cases.aluno import AlunoUseCases
from app.presentation.dependencies import (
    get_aluno_use_cases, requer_professor,
    requer_professor_ou_coordenador,
)
from app.presentation.schemas.aluno_schemas import AlunoCreate,
AlunoResponse

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

@router.post("", response_model=AlunoResponse,
status_code=status.HTTP_201_CREATED)
async def criar_aluno(
    dados: AlunoCreate,
    use_cases: AlunoUseCases = Depends(get_aluno_use_cases),
    _=Depends(requer_professor),
):
    return await use_cases.criar_aluno(AlunoDTO(dados.nome,
dados.turma, dados.curso_id))

@router.get("", response_model=list[AlunoResponse])
async def listar_alunos(
    use_cases: AlunoUseCases = Depends(get_aluno_use_cases),
    _=Depends(requer_professor_ou_coordenador),
):
    return await use_cases.listar_alunos()

@router.get("/{aluno_id}", response_model=AlunoResponse)
async def buscar_aluno(
    aluno_id: int,
```

```
        use_cases: AlunoUseCases = Depends(get_aluno_use_cases),
        _=Depends(requer_professor_ou_coordenador),
    ):
        return await use_cases.buscar_aluno(aluno_id)

@router.put("/{aluno_id}", response_model=AlunoResponse)
async def atualizar_aluno(
    aluno_id: int,
    dados: AlunoCreate,
    use_cases: AlunoUseCases = Depends(get_aluno_use_cases),
    _=Depends(requer_professor),
):
    return await use_cases.atualizar_aluno(
        aluno_id, AlunoDTO(dados.nome, dados.turma, dados.curso_id)
    )

@router.delete("/{aluno_id}", status_code=status.HTTP_204_NO_CONTENT)
async def deletar_aluno(
    aluno_id: int,
    use_cases: AlunoUseCases = Depends(get_aluno_use_cases),
    _=Depends(requer_professor),
):
    await use_cases.deletar_aluno(aluno_id)
    return None
```

Arquivo: "app/presentation/api/curso_router.py"

Python

```
from fastapi import APIRouter, Depends, status

from app.application.dtos.curso import CursoDTO
from app.application.use_cases.curso import CursoUseCases
from app.presentation.dependencies import (
    get_curso_use_cases, requer_coordenador,
    requer_professor_ou_coordenador,
)
from app.presentation.schemas.curso_schemas import CursoCreate,
CursoResponse

router = APIRouter(prefix="/cursos", tags=["Cursos"])
```



```
@router.post("", response_model=CursoResponse,
status_code=status.HTTP_201_CREATED)
async def criar_curso(
    dados: CursoCreate,
    use_cases: CursoUseCases = Depends(get_curso_use_cases),
    _=Depends(requer_coordenador),
):
    return await use_cases.criar_curso(CursoDTO(dados.nome,
dados.duracao))

@router.get("", response_model=list[CursoResponse])
async def listar_cursos(
    use_cases: CursoUseCases = Depends(get_curso_use_cases),
    _=Depends(requer_professor_ou_coordenador),
):
    return await use_cases.listar_cursos()

@router.get("/{curso_id}", response_model=CursoResponse)
async def buscar_curso(
    curso_id: int,
    use_cases: CursoUseCases = Depends(get_curso_use_cases),
    _=Depends(requer_professor_ou_coordenador),
):
    return await use_cases.buscar_curso(curso_id)

@router.put("/{curso_id}", response_model=CursoResponse)
async def atualizar_curso(
    curso_id: int,
    dados: CursoCreate,
    use_cases: CursoUseCases = Depends(get_curso_use_cases),
    _=Depends(requer_coordenador),
):
    return await use_cases.atualizar_curso(curso_id,
CursoDTO(dados.nome, dados.duracao))

@router.delete("/{curso_id}", status_code=status.HTTP_204_NO_CONTENT)
async def deletar_curso(
    curso_id: int,
    use_cases: CursoUseCases = Depends(get_curso_use_cases),
    _=Depends(requer_coordenador),
):
    await use_cases.deletar_curso(curso_id)
```



```
return None
```

Arquivo: "app/presentation/api/disciplina_router.py"

Python

```
from fastapi import APIRouter, Depends, status

from app.application.dtos.disciplina import DisciplinaDTO
from app.application.use_cases.disciplina import DisciplinaUseCases
from app.presentation.dependencies import (
    get_disciplina_use_cases, requer_professor,
    requer_professor_ou_coordenador,
)
from app.presentation.schemas.disciplina_schemas import
DisciplinaCreate, DisciplinaResponse

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

@router.post("", response_model=DisciplinaResponse,
status_code=status.HTTP_201_CREATED)
async def criar_disciplina(
    dados: DisciplinaCreate,
    use_cases: DisciplinaUseCases = Depends(get_disciplina_use_cases),
    _=Depends(requer_professor),
):
    return await use_cases.criar_disciplina(
        DisciplinaDTO(dados.nome, dados.carga_horaria, dados.curso_id)
    )

@router.get("", response_model=list[DisciplinaResponse])
async def listar_disciplinas(
    use_cases: DisciplinaUseCases = Depends(get_disciplina_use_cases),
    _=Depends(requer_professor_ou_coordenador),
):
    return await use_cases.listar_disciplinas()

@router.get("/{disciplina_id}", response_model=DisciplinaResponse)
async def buscar_disciplina(
    disciplina_id: int,
    use_cases: DisciplinaUseCases = Depends(get_disciplina_use_cases),
    _=Depends(requer_professor_ou_coordenador),
```



```
) :  
    return await use_cases.buscar_disciplina(disciplina_id)  
  
@router.put("/{disciplina_id}", response_model=DisciplinaResponse)  
async def atualizar_disciplina(  
    disciplina_id: int,  
    dados: DisciplinaCreate,  
    use_cases: DisciplinaUseCases = Depends(get_disciplina_use_cases),  
    _=Depends(requer_professor),  
):  
    return await use_cases.atualizar_disciplina(  
        disciplina_id, DisciplinaDTO(dados.nome, dados.carga_horaria,  
dados.curso_id)  
    )  
  
@router.delete("/{disciplina_id}",  
status_code=status.HTTP_204_NO_CONTENT)  
async def deletar_disciplina(  
    disciplina_id: int,  
    use_cases: DisciplinaUseCases = Depends(get_disciplina_use_cases),  
    _=Depends(requer_professor),  
):  
    await use_cases.deletar_disciplina(disciplina_id)  
    return None
```

Arquivo: "app/presentation/api/matricula_router.py"

Python

```
from fastapi import APIRouter, Depends, status  
  
from app.application.dtos.matricula import MatriculaDTO  
from app.application.use_cases.matricula import MatriculaUseCases  
from app.presentation.dependencies import (  
    get_matricula_use_cases, requer_coordenador,  
    requer_professor_ou_coordenador,  
)  
from app.presentation.schemas.matricula_schemas import MatriculaCreate,  
MatriculaResponse  
  
router = APIRouter(prefix="/matriculas", tags=["Matriculas"])
```

```
@router.post("", response_model=MatriculaResponse,
status_code=status.HTTP_201_CREATED)
async def criar_matricula(
    dados: MatriculaCreate,
    use_cases: MatriculaUseCases = Depends(get_matricula_use_cases),
    _=Depends(requer_coordenador),
):
    return await use_cases.criar_matricula(MatriculaDTO(dados.aluno_id,
dados.disciplina_id))

@router.get("", response_model=list[MatriculaResponse])
async def listar_matriculas(
    use_cases: MatriculaUseCases = Depends(get_matricula_use_cases),
    _=Depends(requer_professor_ou_coordenador),
):
    return await use_cases.listar_matriculas()

@router.get("/{aluno_id}/{disciplina_id}",
response_model=MatriculaResponse)
async def buscar_matricula(
    aluno_id: int,
    disciplina_id: int,
    use_cases: MatriculaUseCases = Depends(get_matricula_use_cases),
    _=Depends(requer_professor_ou_coordenador),
):
    return await use_cases.buscar_matricula(aluno_id, disciplina_id)

@router.delete("/{aluno_id}/{disciplina_id}",
status_code=status.HTTP_204_NO_CONTENT)
async def deletar_matricula(
    aluno_id: int,
    disciplina_id: int,
    use_cases: MatriculaUseCases = Depends(get_matricula_use_cases),
    _=Depends(requer_coordenador),
):
    await use_cases.deletar_matricula(aluno_id, disciplina_id)
    return None
```

Arquivo: "app/presentation/api/auth_router.py"



Python

```
from fastapi import APIRouter

from app.infrastructure.security.auth_backend import auth_backend
from app.infrastructure.security.user_manager import fastapi_users
from app.presentation.schemas.user_schemas import UserCreate, UserRead

router = APIRouter()
router.include_router(
    fastapi_users.get_auth_router(auth_backend),
    prefix="/auth/jwt", tags=["Auth"]
)

router.include_router(
    fastapi_users.get_register_router(UserRead, UserCreate),
    prefix="/auth", tags=["Auth"]
)
```

Arquivo: "app/presentation/api/__init__.py"

Python

```
# arquivo vazio
```

9) Ajustando o Ponto de Entrada da Aplicação (app/main.py)

Neste momento o arquivo "*main.py*" pode ser simplificado, importando apenas os routers da camada de apresentação.

Arquivo: "app/main.py"

Python

```
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

from app.core.exceptions import ApplicationError
from app.presentation.api.aluno_router import router as aluno_router
from app.presentation.api.auth_router import router as auth_router
from app.presentation.api.curso_router import router as curso_router
from app.presentation.api.disciplina_router import router as
disciplina_router
from app.presentation.api.matricula_router import router as
matricula_router
```

```
app = FastAPI(
    title="Sistema Acadêmico - Modular",
    description="API para gerenciamento de cursos e disciplinas",
    version="1.0.0",
)
app.include_router(auth_router)
app.include_router(aluno_router)
app.include_router(curso_router)
app.include_router(disciplina_router)
app.include_router(matricula_router)

@app.exception_handler(ApplicationError)
async def application_error_handler(request: Request, exc:
ApplicationError):
    return JSONResponse(status_code=exc.status_code, content={"detail":
exc.message})

@app.get("/")
def raiz():
    return {"mensagem": "Bem-vindo à API do Sistema Acadêmico!"}
```

10) Definindo o Arquivo de Configurações (app/core/config.py)

Vamos criar o arquivo "config.py" para centralizar as configurações que antes estavam espalhadas.

Arquivo: "app/core/config.py"

```
Python
import os

DATABASE_URL = os.getenv(
    "DATABASE_URL",
    "postgresql+psycpg://post_user:post_pass@db:5432/fastapi_db",
)
JWT_SECRET = os.getenv("JWT_SECRET",
"SUPER_SECRET_KEY_123_ALTERE_ISS0")
JWT_LIFETIME_SECONDS = 3600
RESET_PASSWORD_TOKEN_SECRET = os.getenv("RESET_PASSWORD_TOKEN_SECRET",
"RESET_PASSWORD_SECRET")
```

```
VERIFICATION_TOKEN_SECRET = os.getenv("VERIFICATION_TOKEN_SECRET",  
"VERIFY_SECRET")
```

11) Revisando os Arquivos de Importação

Para que o Python reconheça as novas pastas como pacotes e as importações funcionem corretamente dentro do Docker, é necessário adicionar arquivos `__init__.py` em cada diretório. Abaixo está a lista completa e na sequência aqueles que requerem conteúdo.

Camada de Domínio (app/domain)

- app/domain/__init__.py (Vazio)
- app/domain/entities/__init__.py
- app/domain/repositories/__init__.py
- app/domain/services/__init__.py

Camada de Aplicação (app/application)

- app/application/__init__.py (Vazio)
- app/application/dtos/__init__.py
- app/application/use_cases/__init__.py

Camada de Infraestrutura (app/infrastructure)

- app/infrastructure/__init__.py (Vazio)
- app/infrastructure/database/__init__.py
- app/infrastructure/repositories/__init__.py
- app/infrastructure/security/__init__.py

Camada de Apresentação (app/presentation)

- app/presentation/__init__.py (Vazio)
- app/presentation/api/__init__.py (Vazio)
- app/presentation/dependencies/__init__.py
- app/presentation/schemas/__init__.py

12) Considerações Finais: Adaptação para Arquitetura Limpa

Utilização do Docker e do Postgres

O uso de *Docker* e *PostgreSQL* se encaixa adequadamente na *Clean Architecture*, pois ambos representam a camada mais externa (Frameworks e Drivers). As

mudanças na estrutura do código não afetam diretamente a configuração do Docker, mas reforçam a ideia de que o banco de dados é um detalhe de infraestrutura.

- *docker-compose.yml*: não precisa de alterações significativas, pois ele já orquestra os serviços da aplicação (web) e do banco de dados (db). A aplicação (web) continuará a se conectar ao db através da DATABASE_URL definida nas variáveis de ambiente.
- *Dockerfile*: também não requer grandes mudanças. Ele continuará a construir a imagem da sua aplicação Python, instalando as dependências do requirements.txt. Foi inserido um único comando, para que as configurações de migração sejam refletidas dentro do container Docker.

Arquivo: "Dockerfile"

Python

```
# Imagem oficial e leve (slim) do Python
FROM python:3.12-slim

# Cria um usuário correspondente ao seu usuário local para evitar
# problemas de permissão
RUN useradd -u 1000 -m appuser

# Define o diretório de trabalho dentro do container
WORKDIR /code

# Garante que a pasta pertence ao novo usuário
RUN chown appuser:appuser /code

# Copia o arquivo de requerimentos primeiro (ajuda no cache do Docker)
COPY ./requirements.txt /code/requirements.txt

# Instala as dependências do Python
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

# Inclui configuração de migrações
COPY ./alembic.ini /code/alembic.ini

# Copia todo o resto do código para dentro do container
COPY ./app /code/app

# Muda para o usuário não-root antes de rodar a aplicação
USER appuser
```

Comando para rodar o servidor Uvicorn

```
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000",  
"--reload"]
```

- Alembic: as migrações do Alembic continuarão a ser gerenciadas na camada de infraestrutura, interagindo com os “*CursoModel*”, “*DisciplinaModel*” e “*UserModel*” definidos em “*app/infrastructure/database/models.py*”.

Etapa de Testes

Uma das grandes vantagens da Clean Architecture é a melhoria na testabilidade. No formato atual, será possível testar as regras de negócio (entidades e casos de uso) de forma isolada, sem a necessidade de um banco de dados real ou do framework FastAPI.

- Testes de Domínio: possibilidade de testar as entidades e serviços de domínio de forma isolada.
- Testes de Aplicação: possibilidade de testar os casos de uso, mockando as interfaces de repositório.
- Testes de Infraestrutura: possibilidade de testar as implementações de repositório, interagindo com um banco de dados de teste (ex: *SQLite* em memória).
- Testes de Apresentação (API): possibilidade de testar os “*endpoints*” do FastAPI, mockando os casos de uso.

Portanto, embora a adaptação da aplicação para Arquitetura Limpa exija uma refatoração inicial significativa, ela trará benefícios a longo prazo em termos de manutenibilidade, testabilidade e escalabilidade. Ao separar as preocupações em camadas bem definidas, a aplicação torna-se mais robusta e adaptável a mudanças futuras, fundamental para o sucesso dos projetos de software.

Este passo a passo tem por objetivo ser um ponto de partida para entender e implementar a Arquitetura Limpa (Clean Architecture). Contudo, é importante salientar que será a prática quem possibilitará a compreensão profunda dos conceitos apresentados.

13) Executando e Testando a Aplicação

SUBINDO APLICAÇÃO E CRIANDO OS USUÁRIOS:

No Terminal de Comandos
(Dentro da Pasta da Aplicação)

Shell

```
docker compose up --build
```

No Terminal de Comandos
Criar Usuário - Papel de Professor (POST)

Shell

```
curl -X POST http://localhost:17000/auth/register \  
-H "Content-Type: application/json" \  
-d '{  
  "email": "professor@gmail.com",  
  "password": "@1234@5678",  
  "role": "professor"  
}'
```

No Terminal de Comandos
Criar Usuário - Papel de Coordenador (POST)

Shell

```
curl -X POST http://localhost:17000/auth/register \  
-H "Content-Type: application/json" \  
-d '{  
  "email": "coordenador@gmail.com",  
  "password": "@1234@5678",  
  "role": "coordenador"  
}'
```

AUTENTICANDO E OBTENDO O TOKEN:

Nesta etapa será necessário, após executar os comandos de login a seguir, copiar o token ("access_token") que será retornado no JSON de resposta do FastAPI. Este token será usado nos passos seguintes.

No Terminal de Comandos
Autenticando Usuário - Papel de Professor (POST)



Shell

```
curl -X POST http://localhost:17000/auth/jwt/login \  
-H "Content-Type: application/x-www-form-urlencoded" \  
-d "username=professor@gmail.com&password=@1234@5678"
```

No Terminal de Comandos

Autenticando Usuário - Papel de Coordenador (POST)

Shell

```
curl -X POST http://localhost:17000/auth/jwt/login \  
-H "Content-Type: application/x-www-form-urlencoded" \  
-d "username=coordenador@gmail.com&password=@1234@5678"
```

TESTANDO AS PERMISSÕES DEFINIDAS:

Para os comandos a seguir, é necessário substituir o texto **TOKEN_PROFESSOR** e **TOKEN_COORDENADOR** pelos hashes reais recebidos nos passos anteriores.

No Terminal de Comandos

Criar Curso - Papel de Professor (POST)

Shell

```
curl -X POST http://localhost:17000/cursos \  
-H "Authorization: Bearer TOKEN_PROFESSOR" \  
-H "Content-Type: application/json" \  
-d '{"nome": "Sistemas de Informação", "duracao": 8}'
```

OBS.: Deve retornar 403 Forbidden / Acesso Negado.

No Terminal de Comandos

Criar Curso - Papel de Coordenador (POST)

Shell

```
curl -X POST http://localhost:17000/cursos \  
-H "Authorization: Bearer TOKEN_COORDENADOR" \  
-H "Content-Type: application/json" \  
-d '{"nome": "Sistemas de Informação", "duracao": 8}'
```

OBS.: Deve retornar 201 Created e gerar o ID = 1.

No Terminal de Comandos

Criar Disciplina - Papel de Professor (POST)



Shell

```
curl -X POST http://localhost:17000/disciplinas \  
-H "Authorization: Bearer TOKEN_PROFESSOR" \  
-H "Content-Type: application/json" \  
-d '{"nome": "Programação Web", "carga_horaria": 80, "curso_id":  
1}'
```

OBS.: Deve retornar 201 Created e gerar o ID = 1.

No Terminal de Comandos

Criar Disciplina - Papel de Coordenador (POST)

Shell

```
curl -X POST http://localhost:17000/disciplinas \  
-H "Authorization: Bearer TOKEN_COORDENADOR" \  
-H "Content-Type: application/json" \  
-d '{"nome": "Programação Web", "carga_horaria": 80, "curso_id":  
1}'
```

OBS.: Deve retornar 403 Forbidden / Acesso Negado.

No Terminal de Comandos

Listar Cursos - Papel de Professor (GET)

Shell

```
curl -X GET http://localhost:17000/cursos \  
-H "Authorization: Bearer TOKEN_PROFESSOR"
```

OBS.: Deve retornar 200 OK com a lista de cursos.

No Terminal de Comandos

Listar Cursos - Papel de Coordenador (GET)

Shell

```
curl -X GET http://localhost:17000/cursos \  
-H "Authorization: Bearer TOKEN_COORDENADOR"
```

OBS.: Deve retornar 200 OK com a lista de cursos.

Arquivo: "app/models/user.py"

Python



INSTITUTO FEDERAL
Paraná
Campus Paranaguá



Ministério da Educação

Arquivo: "app/models/user.py"

Python

Arquivo: "app/models/user.py"

Python