📘 Manual de Implementação de DTOs (Data Transfer Objects)

🎯 Objetivo

Este manual fornece um guia completo para implementar DTOs (Data Transfer Objects) usando Pydantic em projetos Python, seguindo o padrão de excelência do projeto CaseBem. É voltado para projetos que **não possuem nenhuma estrutura de DTOs ou validações implementada**.

📋 Índice

🔍 O que são DTOs?

DTOs (Data Transfer Objects) são objetos simples usados para transferir dados entre camadas da aplicação, especialmente entre a camada de apresentação (APIs, formulários) e a camada de lógica de negócio.

Características dos DTOs:

- ✅ Apenas dados (sem lógica de negócio complexa)
- ✅ Validação automática de tipos e formatos
- ✅ Conversão automática de dados
- ✅ Documentação integrada (schemas JSON/OpenAPI)
- ✅ Reutilizáveis em diferentes contextos

🚀 Por que usar DTOs?

1. Validação Automática

```
# Sem DTO
def criar_usuario(nome: str, email: str, idade: int):
    if not nome or len(nome) < 2:
        raise ValueError("Nome inválido")
    if "@" not in email:
        raise ValueError("Email inválido")
    if idade < 18:
        raise ValueError("Idade inválida")
    # ... mais validações ...

# Com DTO
class UsuarioDTO(BaseDTO):
    nome: str = Field(min_length=2)
    email: EmailStr
    idade: int = Field(ge=18)

# Validação automática!
usuario = UsuarioDTO(nome="João", email="joao@email.com", idade=25)
```

2. Separação de Responsabilidades

- **Model (DB):** Representa dados no banco de dados
- **DTO:** Representa dados da API/formulário
- **Service:** Lógica de negócio

3. Documentação Automática

- Geração automática de schema OpenAPI
- Exemplos de uso integrados
- Validações documentadas

4. Segurança

- Controle preciso de quais campos podem ser recebidos
- Prevenção de mass assignment
- Sanitização automática de dados

Arquitetura Proposta

```
seu-projeto/
├── dtos/
│   ├── __init__.py          # Imports centralizados
│   ├── base_dto.py          # Classe base para todos os DTOs
│   ├── usuario_dtos.py      # DTOs relacionados a usuários
│   ├── produto_dtos.py      # DTOs relacionados a produtos
│   └── pedido_dtos.py       # DTOs relacionados a pedidos
│
├── util/
│   └── validacoes_dto.py    # Funções de validação reutilizáveis
│
└── model/
    └── usuario_model.py      # Models do banco de dados
```

Princípios da Arquitetura:

1. **Um arquivo por domínio:** Agrupe DTOs relacionados
2. **BaseDTO:** Classe base com configurações comuns
3. **Validações centralizadas:** Reutilize funções de validação
4. **Imports facilitados:** Use `__init__.py` para simplificar imports

Passo a Passo de Implementação

PASSO 1: Instalar Dependências

```
pip install pydantic[email]
```

Adicione ao `requirements.txt`:

```
pydantic>=2.0.0
email-validator>=2.0.0
```

PASSO 2: Criar Estrutura de Diretórios

```
mkdir -p dtos
mkdir -p util
touch dtos/__init__.py
touch dtos/base_dto.py
touch util/validacoes_dto.py
```

PASSO 3: Implementar Exceção Personalizada

Arquivo: `util/validacoes_dto.py`

```
"""
Biblioteca centralizada de validações para DTOs
"""

class ValidacaoError(ValueError):
    """Exceção personalizada para erros de validação"""
    pass
```

PASSO 4: Criar Funções de Validação

Adicione ao arquivo `util/validacoes_dto.py`:

```
import re
from typing import Optional
from decimal import Decimal

def validar_texto_obrigatorio(
    texto: str,
    campo: str = "Campo",
    min_chars: int = 1,
    max_chars: int = 255
) -> str:
    """
    Valida texto obrigatório com limites de tamanho

    Args:
        texto: Texto a ser validado
        campo: Nome do campo (para mensagens de erro)
        min_chars: Tamanho mínimo
        max_chars: Tamanho máximo

    Returns:
        Texto validado e limpo

    Raises:
        ValidacaoError: Se validação falhar
    """
    if not texto or not texto.strip():
        raise ValidacaoError(f'{campo} é obrigatório')

    texto_limpo = texto.strip()

    if len(texto_limpo) < min_chars:
        raise ValidacaoError(f'{campo} deve ter pelo menos {min_chars} caracteres')

    if len(texto_limpo) > max_chars:
        raise ValidacaoError(f'{campo} deve ter no máximo {max_chars} caracteres')

    return texto_limpo


def validar_texto_opcional(
    texto: Optional[str],
    max_chars: int = 500
) -> Optional[str]:
    """
    Valida texto opcional

    Args:
        texto: Texto a ser validado (pode ser None)
        max_chars: Tamanho máximo

    Returns:
```

```
        Texto validado ou None

    Raises:
        ValidacaoError: Se texto exceder tamanho máximo
    """
    if not texto or not texto.strip():
        return None

    texto_limpo = texto.strip()

    if len(texto_limpo) > max_chars:
        raise ValidacaoError(f'Texto deve ter no máximo {max_chars} caracteres')

    return texto_limpo


def validar_cpf(cpf: Optional[str]) -> Optional[str]:
    """
    Valida CPF brasileiro com dígitos verificadores

    Args:
        cpf: CPF a ser validado (pode conter máscaras)

    Returns:
        CPF limpo (apenas números) ou None se vazio

    Raises:
        ValidacaoError: Se CPF for inválido
    """
    if not cpf:
        return None

    # Remover caracteres especiais
    cpf_limpo = re.sub(r'[^0-9]', '', cpf)

    if len(cpf_limpo) != 11:
        raise ValidacaoError('CPF deve ter 11 dígitos')

    # Verificar se todos os dígitos são iguais
    if cpf_limpo == cpf_limpo[0] * 11:
        raise ValidacaoError('CPF inválido')

    # Validar dígito verificador
    def calcular_digito(cpf_parcial):
        soma = sum(int(cpf_parcial[i]) * (len(cpf_parcial) + 1 - i)
                   for i in range(len(cpf_parcial)))
        resto = soma % 11
        return 0 if resto < 2 else 11 - resto

    if int(cpf_limpo[9]) != calcular_digito(cpf_limpo[:9]):
        raise ValidacaoError('CPF inválido')

    if int(cpf_limpo[10]) != calcular_digito(cpf_limpo[:10]):
        raise ValidacaoError('CPF inválido')

    return cpf_limpo


def validar_telefone(telefone: str) -> str:
    """
    Valida telefone brasileiro (celular ou fixo)

    Args:
        telefone: Telefone a ser validado

    Returns:
```

```
    Returns:
        Telefone limpo (apenas números)

    Raises:
        ValidacaoError: Se telefone for inválido
    """
    if not telefone:
        raise ValidacaoError('Telefone é obrigatório')

    # Remover caracteres especiais
    telefone_limpo = re.sub(r'[^0-9]', '', telefone)

    # Telefone deve ter 10 (fixo) ou 11 (celular) dígitos
    if len(telefone_limpo) not in [10, 11]:
        raise ValidacaoError('Telefone deve ter 10 ou 11 dígitos')

    # Validar DDD (11 a 99)
    ddd = int(telefone_limpo[:2])
    if ddd < 11 or ddd > 99:
        raise ValidacaoError('DDD inválido')

    return telefone_limpo


def validar_valor_monetario(
    valor: Optional[Decimal],
    campo: str = "Valor",
    obrigatorio: bool = True,
    min_valor: Optional[Decimal] = None
) -> Optional[Decimal]:
    """
    Valida valor monetário

    Args:
        valor: Valor a ser validado
        campo: Nome do campo
        obrigatorio: Se o valor é obrigatório
        min_valor: Valor mínimo permitido

    Returns:
        Valor validado

    Raises:
        ValidacaoError: Se validação falhar
    """
    if valor is None:
        if obrigatorio:
            raise ValidacaoError(f'{campo} é obrigatório')
        return None

    if not isinstance(valor, Decimal):
        try:
            valor = Decimal(str(valor))
        except:
            raise ValidacaoError(f'{campo} deve ser um valor numérico válido')

    if min_valor is not None and valor < min_valor:
        raise ValidacaoError(f'{campo} deve ser maior ou igual a {min_valor}')

    return valor


def validar_enum_valor(valor: any, enum_class, campo: str = "Campo"):
    """
    Valida se valor está em um enum
```

```
    Args:
        valor: Valor a ser validado
        enum_class: Classe do enum
        campo: Nome do campo

    Returns:
        Valor do enum validado

    Raises:
        ValidacaoError: Se valor não estiver no enum
    """
    if isinstance(valor, str):
        try:
            return enum_class(valor.upper())
        except ValueError:
            valores_validos = [item.value for item in enum_class]
            raise ValidacaoError(
                f'{campo} deve ser uma das opções: {", ".join(valores_validos)}'
            )

    if valor not in enum_class:
        valores_validos = [item.value for item in enum_class]
        raise ValidacaoError(
            f'{campo} deve ser uma das opções: {", ".join(valores_validos)}'
        )

    return valor
```

PASSO 5: Criar ValidadorWrapper

Adicione ao final do arquivo `util/validacoes_dto.py`:

```
class ValidadorWrapper:
    """
    Classe para facilitar o uso de validadores em field_validators.
    Reduz código repetitivo e padroniza tratamento de erros.
    """

    @staticmethod
    def criar_validador(funcao_validacao, campo_nome: str = None, **kwargs):
        """
        Cria um validador pronto para usar com @field_validator.

        Args:
            funcao_validacao: Função de validação a ser chamada
            campo_nome: Nome do campo para mensagens de erro
            **kwargs: Argumentos adicionais para a função

        Returns:
            Função validador pronta para usar

        Exemplo:
            validar_nome = ValidadorWrapper.criar_validador(
                validar_texto_obrigatorio, "Nome", min_chars=2, max_chars=100
            )
        """
        def validador(valor):
            try:
                if campo_nome:
                    return funcao_validacao(valor, campo_nome, **kwargs)
                else:
                    return funcao_validacao(valor, **kwargs)
            except ValidacaoError as e:
                raise ValueError(str(e))
        return validador
```

PASSO 6: Criar BaseDTO

Arquivo: `dtos/base_dto.py`

```
"""
Classe base para todos os DTOs do sistema.
Fornece configurações padrão e métodos de validação comuns.
"""

from pydantic import BaseModel, ConfigDict
from typing import Dict, Any
from util.validacoes_dto import ValidacaoError


class BaseDTO(BaseModel):
    """
    Classe base para todos os DTOs do sistema.
    Fornece configurações padrão e métodos de validação comuns.

    Esta classe implementa:
    - Configurações padrão do Pydantic
    - Wrapper para tratamento de erros de validação
    - Métodos auxiliares para conversão de dados
    """

    model_config = ConfigDict(
        # Remover espaços em branco automaticamente
        str_strip_whitespace=True,
        # Validar na atribuição também (não só na criação)
```

```
        validate_assignment=True,
        # Usar valores dos enums ao invés dos objetos
        use_enum_values=True,
        # Permitir population by name (útil para formulários HTML)
        populate_by_name=True,
        # Validar valores padrão também
        validate_default=True
    )

    @classmethod
    def criar_exemplo_json(cls, **overrides) -> Dict[str, Any]:
        """
        Cria um exemplo JSON para documentação da API.
        Pode ser sobrescrito nas classes filhas.

        Args:
            **overrides: Valores específicos para sobrescrever no exemplo

        Returns:
            Dict com exemplo de dados para este DTO
        """
        return {"exemplo": "Sobrescrever na classe filha", **overrides}

    @classmethod
    def validar_campo_wrapper(cls, validador_func, campo_nome: str = ""):
        """
        Wrapper para padronizar o tratamento de erros de validação.
        Evita repetir try/except em cada field_validator.

        Args:
            validador_func: Função de validação a ser envolvida
            campo_nome: Nome do campo para mensagens de erro

        Returns:
            Função wrapper que trata os erros automaticamente
        """
        def wrapper(valor, **kwargs):
            try:
                if campo_nome:
                    return validador_func(valor, campo_nome, **kwargs)
                else:
                    return validador_func(valor, **kwargs)
            except ValidacaoError as e:
                raise ValueError(str(e))
        return wrapper

    def to_dict(self) -> dict:
        """
        Converte DTO para dicionário simples.
        Remove campos None para limpar o retorno.

        Returns:
            Dicionário com os dados do DTO
        """
        return self.model_dump(exclude_none=True)

    def to_json(self) -> str:
        """
        Converte DTO para JSON.
        Remove campos None para limpar o retorno.

        Returns:
            String JSON com os dados do DTO
        """
        return self.model_dump_json(exclude_none=True)
```

```
    @classmethod
    def from_dict(cls, data: dict):
        """
        Cria DTO a partir de dicionário.

        Args:
            data: Dicionário com os dados

        Returns:
            Instância do DTO
        """
        return cls(**data)

    def __str__(self) -> str:
        """Representação string melhorada do DTO"""
        campos = ', '.join([f"{k}={v}" for k, v in self.to_dict().items()])
        return f"{self.__class__.__name__}({campos})"

    def __repr__(self) -> str:
        """Representação técnica do DTO"""
        return self.__str__()
```

PASSO 7: Criar Primeiro DTO por Domínio

Arquivo: `dtos/usuario_dtos.py`

```
 """
DTOs relacionados a usuários.
Agrupa todas as validações e estruturas de dados para operações com usuários.
"""

from pydantic import EmailStr, Field, field_validator
from typing import Optional
from .base_dto import BaseDTO
from util.validacoes_dto import (
    validar_texto_obrigatorio, validar_cpf, validar_telefone
)


class CriarUsuarioDTO(BaseDTO):
    """
    DTO para criação de novo usuário.
    Usado em formulários de registro.
    """

    nome: str = Field(
        ...,
        min_length=2,
        max_length=100,
        description="Nome completo do usuário"
    )
    email: EmailStr = Field(
        ...,
        description="E-mail válido do usuário"
    )
    telefone: str = Field(
        ...,
        min_length=10,
        description="Telefone com DDD"
    )
    cpf: Optional[str] = Field(
        None,
        description="CPF (opcional)"
```

```
    )

    @field_validator('nome')
    @classmethod
    def validar_nome(cls, v: str) -> str:
        validador = cls.validar_campo_wrapper(
            lambda valor, campo: validar_texto_obrigatorio(
                valor, campo, min_chars=2, max_chars=100
            ),
            "Nome"
        )
        return validador(v)

    @field_validator('cpf')
    @classmethod
    def validar_cpf_campo(cls, v: Optional[str]) -> Optional[str]:
        if not v:
            return v
        validador = cls.validar_campo_wrapper(
            lambda valor, campo: validar_cpf(valor),
            "CPF"
        )
        return validador(v)

    @field_validator('telefone')
    @classmethod
    def validar_telefone_campo(cls, v: str) -> str:
        validador = cls.validar_campo_wrapper(
            lambda valor, campo: validar_telefone(valor),
            "Telefone"
        )
        return validador(v)

    @classmethod
    def criar_exemplo_json(cls, **overrides) -> dict:
        """Exemplo de dados para documentação da API"""
        exemplo = {
            "nome": "João Silva",
            "email": "joao.silva@email.com",
            "telefone": "(11) 99999-9999",
            "cpf": "123.456.789-01"
        }
        exemplo.update(overrides)
        return exemplo


class AtualizarUsuarioDTO(BaseDTO):
    """
    DTO para atualização de dados do usuário.
    Campos opcionais para atualização parcial.
    """

    nome: Optional[str] = Field(
        None,
        min_length=2,
        max_length=100,
        description="Nome completo"
    )
    telefone: Optional[str] = Field(
        None,
        description="Telefone"
    )

    @field_validator('nome')
    @classmethod
```

```
    def validar_nome(cls, v: Optional[str]) -> Optional[str]:
        if not v:
            return v
        validador = cls.validar_campo_wrapper(
            lambda valor, campo: validar_texto_obrigatorio(
                valor, campo, min_chars=2, max_chars=100
            ),
            "Nome"
        )
        return validador(v)

    @field_validator('telefone')
    @classmethod
    def validar_telefone_campo(cls, v: Optional[str]) -> Optional[str]:
        if not v:
            return v
        validador = cls.validar_campo_wrapper(
            lambda valor, campo: validar_telefone(valor),
            "Telefone"
        )
        return validador(v)


# Configurar exemplos JSON nos model_config
CriarUsuarioDTO.model_config.update({
    "json_schema_extra": {
        "example": CriarUsuarioDTO.criar_exemplo_json()
    }
})
```

PASSO 8: Configurar Imports Facilitados

Arquivo: `dtos/__init__.py`

```
"""
Pacote de DTOs do sistema.

Este módulo centraliza todos os DTOs (Data Transfer Objects) organizados por funcionalidade:
- BaseDTO: Classe base com configurações comuns
- usuario_dtos: DTOs relacionados a usuários

Imports facilitados para os DTOs mais comuns:
"""

# Base
from .base_dto import BaseDTO

# Usuário
from .usuario_dtos import (
    CriarUsuarioDTO,
    AtualizarUsuarioDTO
)

__all__ = [
    # Base
    'BaseDTO',

    # Usuário
    'CriarUsuarioDTO',
    'AtualizarUsuarioDTO',
]
```

PASSO 9: Usar DTOs em Rotas/Controllers

Exemplo com FastAPI:

```
 from fastapi import APIRouter, HTTPException
from dtos import CriarUsuarioDTO, AtualizarUsuarioDTO
from pydantic import ValidationError

router = APIRouter()

@router.post("/usuarios")
def criar_usuario(usuario_dto: CriarUsuarioDTO):
    """
    Cria um novo usuário.

    A validação é automática! Se os dados forem inválidos,
    FastAPI retorna 422 automaticamente.
    """
    try:
        # Converter DTO para dict
        dados = usuario_dto.to_dict()

        # Salvar no banco de dados
        # usuario_service.criar(dados)

        return {"mensagem": "Usuário criado com sucesso", "dados": dados}

    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))


@router.put("/usuarios/{id}")
def atualizar_usuario(id: int, usuario_dto: AtualizarUsuarioDTO):
    """
    Atualiza dados de um usuário.
    """
    try:
        dados = usuario_dto.to_dict()

        # Atualizar no banco de dados
        # usuario_service.atualizar(id, dados)

        return {"mensagem": "Usuário atualizado com sucesso"}

    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
```

Exemplo com Flask:

```python
from flask import Blueprint, request, jsonify
from dtos import CriarUsuarioDTO
from pydantic import ValidationError

usuario_bp = Blueprint('usuario', __name__)

@usuario_bp.route('/usuarios', methods=['POST'])
def criar_usuario():
    """
    Cria um novo usuário.
    """
    try:
        # Validar dados com DTO
        usuario_dto = CriarUsuarioDTO(**request.json)

        # Converter para dict
        dados = usuario_dto.to_dict()

        # Salvar no banco de dados
        # usuario_service.criar(dados)

        return jsonify({
            "mensagem": "Usuário criado com sucesso",
            "dados": dados
        }), 201

    except ValidationError as e:
        return jsonify({"erros": e.errors()}), 422

    except Exception as e:
        return jsonify({"erro": str(e)}), 400
```

Exemplos Práticos

Exemplo 1: DTO Simples com Validações Básicas

```python
from pydantic import Field
from .base_dto import BaseDTO

class ProdutoDTO(BaseDTO):
    """DTO para cadastro de produto"""

    nome: str = Field(..., min_length=3, max_length=100)
    preco: float = Field(..., gt=0)
    estoque: int = Field(..., ge=0)
    ativo: bool = Field(default=True)
```

Exemplo 2: DTO com Enum

```
from enum import Enum
from pydantic import Field, field_validator
from .base_dto import BaseDTO
from util.validacoes_dto import validar_enum_valor

class StatusPedido(str, Enum):
    PENDENTE = "PENDENTE"
    PROCESSANDO = "PROCESSANDO"
    ENVIADO = "ENVIADO"
    ENTREGUE = "ENTREGUE"
    CANCELADO = "CANCELADO"

class PedidoDTO(BaseDTO):
    """DTO para pedido"""

    cliente_id: int = Field(..., gt=0)
    status: StatusPedido = Field(default=StatusPedido.PENDENTE)
    observacoes: Optional[str] = Field(None, max_length=500)

    @field_validator('status')
    @classmethod
    def validar_status(cls, v):
        validador = cls.validar_campo_wrapper(
            lambda valor, campo: validar_enum_valor(valor, StatusPedido, campo),
            "Status"
        )
        return validador(v)
```

Exemplo 3: DTO com Validação Customizada

```
from pydantic import Field, field_validator, ValidationInfo
from .base_dto import BaseDTO

class AlterarSenhaDTO(BaseDTO):
    """DTO para alteração de senha"""

    senha_atual: str = Field(..., min_length=1)
    nova_senha: str = Field(..., min_length=8)
    confirmar_senha: str = Field(..., min_length=8)

    @field_validator('nova_senha')
    @classmethod
    def validar_nova_senha(cls, v: str, info: ValidationInfo) -> str:
        # Validar que nova senha é diferente da atual
        if 'senha_atual' in info.data and v == info.data['senha_atual']:
            raise ValueError('Nova senha deve ser diferente da atual')

        # Validar força da senha
        if not any(c.isupper() for c in v):
            raise ValueError('Senha deve conter pelo menos uma letra maiúscula')

        if not any(c.isdigit() for c in v):
            raise ValueError('Senha deve conter pelo menos um número')

        return v

    @field_validator('confirmar_senha')
    @classmethod
    def senhas_devem_coincidir(cls, v: str, info: ValidationInfo) -> str:
        if 'nova_senha' in info.data and v != info.data['nova_senha']:
            raise ValueError('Senhas não coincidem')
        return v
```

Exemplo 4: DTO para Filtros de Listagem

```
 from typing import Optional
from pydantic import Field
from .base_dto import BaseDTO

class ProdutoFiltroDTO(BaseDTO):
    """DTO para filtros de listagem de produtos"""

    nome_busca: Optional[str] = Field(None, max_length=100)
    preco_min: Optional[float] = Field(None, ge=0)
    preco_max: Optional[float] = Field(None, ge=0)
    ativo: Optional[bool] = None
    categoria_id: Optional[int] = Field(None, gt=0)

    # Paginação
    pagina: int = Field(default=1, ge=1)
    tamanho_pagina: int = Field(default=20, ge=1, le=100)
```

Exemplo 5: DTO com Relacionamentos

```
 from typing import List, Optional
from pydantic import Field
from .base_dto import BaseDTO

class ItemPedidoDTO(BaseDTO):
    """DTO para item do pedido"""

    produto_id: int = Field(..., gt=0)
    quantidade: int = Field(..., gt=0)
    preco_unitario: float = Field(..., gt=0)

class CriarPedidoDTO(BaseDTO):
    """DTO para criação de pedido com itens"""

    cliente_id: int = Field(..., gt=0)
    itens: List[ItemPedidoDTO] = Field(..., min_length=1)
    observacoes: Optional[str] = Field(None, max_length=500)

    @field_validator('itens')
    @classmethod
    def validar_itens(cls, v: List[ItemPedidoDTO]) -> List[ItemPedidoDTO]:
        if not v:
            raise ValueError('Pedido deve ter pelo menos um item')
        return v
```

✅ Boas Práticas

1. Organize DTOs por Domínio

```
# ❌ ERRADO - Um arquivo por DTO
dtos/
├── criar_usuario_dto.py
├── atualizar_usuario_dto.py
├── criar_produto_dto.py
└── atualizar_produto_dto.py

# ✅ CORRETO - Agrupe por domínio
dtos/
├── usuario_dtos.py      # Todos os DTOs de usuário
├── produto_dtos.py      # Todos os DTOs de produto
└── pedido_dtos.py       # Todos os DTOs de pedido
```

2. Use Nomes Descritivos

```
# ❌ ERRADO
class UsuarioDTO(BaseDTO):  # Muito genérico
    pass

# ✅ CORRETO
class CriarUsuarioDTO(BaseDTO):      # Indica ação e contexto
    pass

class AtualizarUsuarioDTO(BaseDTO):  # Claro e específico
    pass

class UsuarioFiltroDTO(BaseDTO):     # Indica propósito
    pass
```

3. Documente seus DTOs

```
class CriarProdutoDTO(BaseDTO):
    """
    DTO para criação de novo produto.
    Usado em formulários de cadastro de produtos.

    Validações:
    - Nome: 3-100 caracteres
    - Preço: Deve ser maior que zero
    - Estoque: Não pode ser negativo
    """

    nome: str = Field(..., min_length=3, max_length=100, description="Nome do produto")
    preco: float = Field(..., gt=0, description="Preço unitário (deve ser > 0)")
    estoque: int = Field(..., ge=0, description="Quantidade em estoque")
```

4. Crie Exemplos JSON

```
class ProdutoDTO(BaseDTO):
    nome: str
    preco: float

    @classmethod
    def criar_exemplo_json(cls, **overrides) -> dict:
        exemplo = {
            "nome": "Notebook Dell",
            "preco": 3500.00
        }
        exemplo.update(overrides)
        return exemplo

# Configurar no model_config
ProdutoDTO.model_config.update({
    "json_schema_extra": {
        "example": ProdutoDTO.criar_exemplo_json()
    }
})
```

5. Reutilize Validações

```
# ❌ ERRADO - Repetir validação em cada DTO
class UsuarioDTO(BaseDTO):
    cpf: str

    @field_validator('cpf')
    @classmethod
    def validar_cpf(cls, v):
        # ... código de validação repetido ...
        pass

# ✅ CORRETO - Usar função centralizada
from util.validacoes_dto import validar_cpf

class UsuarioDTO(BaseDTO):
    cpf: str

    @field_validator('cpf')
    @classmethod
    def validar_cpf_campo(cls, v):
        validador = cls.validar_campo_wrapper(
            lambda valor, campo: validar_cpf(valor),
            "CPF"
        )
        return validador(v)
```

6. Separe DTOs de Models

```
# Model (banco de dados)
class Usuario:
    id: int
    nome: str
    email: str
    senha_hash: str  # Senha criptografada
    criado_em: datetime
    atualizado_em: datetime

# DTO (API/Formulário)
class CriarUsuarioDTO(BaseDTO):
    nome: str
    email: EmailStr
    senha: str  # Senha em texto plano (será criptografada)
    # Não expõe dados internos como senha_hash ou timestamps
```

7. Use Tipos Adequados

```
from pydantic import EmailStr, HttpUrl, conint, condecimal
from datetime import date, datetime
from decimal import Decimal

class ExemploDTO(BaseDTO):
    # ✅ Use EmailStr para emails
    email: EmailStr

    # ✅ Use HttpUrl para URLs
    website: HttpUrl

    # ✅ Use Decimal para valores monetários
    preco: Decimal

    # ✅ Use date/datetime para datas
    data_nascimento: date
    data_cadastro: datetime

    # ✅ Use constrained types para validações específicas
    idade: conint(ge=18, le=120)  # Entre 18 e 120
    desconto: condecimal(ge=0, le=100)  # Entre 0 e 100
```

8. Trate Campos Opcionais Corretamente

```
from typing import Optional

class ProdutoDTO(BaseDTO):
    # Campo obrigatório
    nome: str = Field(...)

    # Campo opcional com valor padrão
    ativo: bool = Field(default=True)

    # Campo opcional sem valor padrão
    descricao: Optional[str] = Field(None)

    # Validar apenas se campo foi fornecido
    @field_validator('descricao')
    @classmethod
    def validar_descricao(cls, v: Optional[str]) -> Optional[str]:
        if v is None:
            return v
        # Validar apenas se não for None
        if len(v) < 10:
            raise ValueError('Descrição muito curta')
        return v
```

Troubleshooting

Problema 1: "ValidationError: field required"

Causa: Campo obrigatório não foi fornecido

Solução:

```
# Certifique-se de que campos obrigatórios usam ... (Ellipsis)
class UsuarioDTO(BaseDTO):
    nome: str = Field(...)  # Obrigatório
    email: EmailStr = Field(...)  # Obrigatório
```

Problema 2: "ImportError: cannot import name 'ValidacaoError'"

Causa: Módulo de validações não encontrado

Solução:

```
# Verifique o caminho correto para o módulo
from util.validacoes_dto import ValidacaoError  # Ajuste o caminho conforme sua estrutura
```

Problema 3: Validação não está sendo executada

Causa: Esqueceu de usar @field_validator

Solução:

```
class UsuarioDTO(BaseDTO):
    cpf: str

    # ✅ Adicione o decorator
    @field_validator('cpf')
    @classmethod
    def validar_cpf(cls, v):
        # ... validação ...
        pass
```

Problema 4: "ValueError: Extra inputs are not permitted"

Causa: DTO recebeu campos não declarados

Solução:

```
# Opção 1: Permitir campos extras (não recomendado)
class UsuarioDTO(BaseDTO):
    model_config = ConfigDict(extra='allow')

# Opção 2: Ignorar campos extras (recomendado)
class UsuarioDTO(BaseDTO):
    model_config = ConfigDict(extra='ignore')

# Opção 3: Proibir campos extras (padrão, mais seguro)
class UsuarioDTO(BaseDTO):
    model_config = ConfigDict(extra='forbid')
```

Problema 5: Enum não aceita valores

Causa: Valor enviado não corresponde aos valores do Enum

Solução:

```
from enum import Enum

class StatusEnum(str, Enum):
    ATIVO = "ATIVO"
    INATIVO = "INATIVO"

# ⍰ Configure use_enum_values=True no model_config
class ProdutoDTO(BaseDTO):
    status: StatusEnum

    model_config = ConfigDict(use_enum_values=True)
```

Problema 6: Validação de senha não compara com outro campo

Causa: Não está usando ValidationInfo

Solução:

```
from pydantic import field_validator, ValidationInfo

class AlterarSenhaDTO(BaseDTO):
    nova_senha: str
    confirmar_senha: str

    @field_validator('confirmar_senha')
    @classmethod
    def senhas_coincidem(cls, v: str, info: ValidationInfo) -> str:
        if 'nova_senha' in info.data and v != info.data['nova_senha']:
            raise ValueError('Senhas não coincidem')
        return v
```

⍰ Recursos Adicionais

Documentação Oficial:

- **Pydantic:** https://docs.pydantic.dev/
- **FastAPI:** https://fastapi.tiangolo.com/
- **Python Type Hints:** https://docs.python.org/3/library/typing.html

Exemplos de Validações Comuns:

- CPF/CNPJ: https://github.com/brazilians/validators
- CEP: https://pycep-correios.readthedocs.io/

- Telefone: https://github.com/daviddrysdale/python-phonenumbers

Padrões de Projeto:

- **DTO Pattern:** https://martinfowler.com/eaaCatalog/dataTransferObject.html
- **Validation Pattern:** https://refactoring.guru/design-patterns/specification

---

✅ Checklist de Implementação

- [ ] Instalar Pydantic (`pip install pydantic[email]`)
- [ ] Criar estrutura de diretórios (`dtos/` , `util/`)
- [ ] Implementar `ValidacaoError` em `util/validacoes_dto.py`
- [ ] Criar funções de validação em `util/validacoes_dto.py`
- [ ] Implementar `ValidadorWrapper` em `util/validacoes_dto.py`
- [ ] Criar `BaseDTO` em `dtos/base_dto.py`
- [ ] Criar primeiro DTO por domínio (ex: `usuario_dtos.py`)
- [ ] Configurar `dtos/__init__.py` com imports facilitados
- [ ] Usar DTOs nas rotas/controllers
- [ ] Testar validações com dados inválidos
- [ ] Documentar DTOs com docstrings
- [ ] Criar exemplos JSON para documentação da API

🎉 Conclusão

Este manual fornece tudo que você precisa para implementar DTOs de forma profissional em seu projeto Python. Seguindo este padrão, você terá:

✅ **Validação automática** de todos os dados de entrada ✅ **Código organizado** e fácil de manter ✅ **Documentação automática** da API ✅ **Segurança** contra dados inválidos ✅ **Reutilização** de código através de funções centralizadas

Próximos passos sugeridos:

1. Implemente a estrutura base (Passos 1-6)
2. Crie seu primeiro DTO seguindo os exemplos
3. Teste com dados válidos e inválidos
4. Expanda gradualmente para outros domínios do sistema

Dúvidas? Consulte os exemplos práticos e o troubleshooting neste manual.

Baseado no projeto CaseBem - Sistema de gerenciamento de casamentos **Última atualização:** 2025 **Versão:** 1.0