Código do Projeto DEV Platform

Conteúdo da pasta: ./src/stake_file

Gerado em: 06/06/2025 às 23:19

Índice

1. Arquivo: .env

```
# ./.env
DB_USER_REMOTE="root"
DB_PASSWORD_REMOTE="Malato%2301"
DB_HOST_REMOTE="127.0.0.1:3306"
DB_NAME_REMOTE="env_management_db"
```

2. Arquivo: .env.development

```
# ./.env.development
ENVIRONMENT=development
DATABASE_URL=mysql+aiomysql://root:Malato%2301@127.0.0.1:3306/user_management

DB_POOL_SIZE=5
DB_MAX_OVERFLOW=10
DB_ECHO=True
LOG_LEVEL=DEBUG DEBUG=True
```

3. Arquivo: .env.production

```
# ./.env.production
# Nota: Este arquivo contém as variáveis de ambiente para o ambiente de produ
ção.
# Certifique-se de que este arquivo não seja incluído no controle de versão,
pois contém informações sensíveis.
# Essas variáveis de ambiente estão em fase de trasferência para melhor gestã
o de segredos em produção e para
# otimizar a validação de dados, visando maior segurança e eficiência.
ENVIRONMENT=production
DATABASE_URL=mysql+aiomysql://root:Malato%2301@127.0.0.1:3306/user_management

DB_POOL_SIZE=20
DB_MAX_OVERFLOW=30
DB_ECHO=False
LOG_LEVEL=INFO
DEBUG=False

# Configurações de validação
VALIDATION_ENABLE_PROFANITY_FILTER=True
VALIDATION_FORBIDDEN_WORDS="palavrao1,palavrao2, termo_proibido" # Lista de p
alavras proibidas, separadas por vírgula
VALIDATION_ALLOWED_DOMAINS="dominio1.com, dominio2.com" # Lista de domínios p
ermitidos, separadas por vírgula
VALIDATION_BUSINESS_HOURS_ONLY=False
```

4. Arquivo: .env.test

```
# ./.env.test
ENVIRONMENT=test
DATABASE_URL=mysql+aiomysql://root:Malato%2301@127.0.0.1:3306/user_management
DB_POOL_SIZE=1
DB_MAX_OVERFLOW=0
DB_ECHO=False
LOG_LEVEL=WARNING
DEBUG=False
```

5. Arquivo: .gitignore

```
# ./.gitignore
# NOTA
# Commit do arquivo poetry.lock permite criar um ambiente determinístico

# Ambientes virtuais
.venv/
venv/
ENV/
env/

# Arquivos de cache Python
__pycache__/
*.py[cod]
*$py.class
.pytest_cache/
.coverage
htmlcov/
.tox/
.nox/

# Distribuição / empacotamento
dist/
build/
*.egg-info/
*.egg

# Documentação gerada pelo MkDocs
site/

# Poetry
poetry.lock
# Descomente a linha acima se NÃO quiser versionar o poetry.lock

# Arquivos de log
*.log
logs/

# Arquivos temporários
*.tmp
*.bak
*.swp
*~

# Arquivos do sistema operacional
.DS_Store
Thumbs.db

# IDEs e editores
.idea/
.vscode/
*.sublime-project
*.sublime-workspace
.project
.pydevproject
.spyderproject
.spyproject
.ropeproject

# Jupyter Notebook
.ipynb_checkpoints

# Arquivos de ambiente
.env
```

```
.env.local
.env.development.local
.env.test.local
.env.production.local
.env.development
.env.test
.env.production

# Diretórios de mídia/uploads (se aplicável)
media/
uploads/

# SQLite DB (se aplicável)
*.sqlite3
*.db

# Arquivos de configuração local
local_settings.py
```

6. Arquivo: alembic.ini

```
# ./alembic.ini
# A generic, single database configuration.

[alembic]
# path to migration scripts
# Use forward slashes (/) also on windows to provide an os agnostic path
script_location = migrations

# template used to generate migration file names; The default value is %%(rev
)s_%%(slug)s
# Uncomment the line below if you want the files to be prepended with date an
d time
# see https://alembic.sqlalchemy.org/en/latest/tutorial.html#editing-the-ini-
file
# for all available tokens
# file_template = %%(year)d_%%(month).2d_%%(day).2d_%%(hour).2d%%(minute).2d-
%%(rev)s_%%(slug)s

# sys.path path, will be prepended to sys.path if present.
# defaults to the current working directory.
prepend_sys_path = .

# timezone to use when rendering the date within the migration file
# as well as the filename.
# If specified, requires the python>=3.9 or backports.zoneinfo library and tz
data library.
# Any required deps can installed by adding `alembic[tz]` to the pip requirem
ents
# string value is passed to ZoneInfo()
# leave blank for localtime
# timezone =

# max length of characters to apply to the "slug" field
# truncate_slug_length = 40

# set to 'true' to run the environment during
# the 'revision' command, regardless of autogenerate
# revision_environment = false

# set to 'true' to allow .pyc and .pyo files without
# a source .py file to be detected as revisions in the
# versions/ directory
# sourceless = false

# version location specification; This defaults
# to migrations/versions.  When using multiple version
# directories, initial revisions must be specified with --version-path.
# The path separator used here should be the separator specified by "version_
path_separator" below.
# version_locations = %(here)s/bar:%(here)s/bat:migrations/versions

# version path separator; As mentioned above, this is the character used to s
plit
# version_locations. The default within new alembic.ini files is "os", which
uses os.pathsep.
# If this key is omitted entirely, it falls back to the legacy behavior of sp
litting on spaces and/or commas.
# Valid values for version_path_separator are:
#
# version_path_separator = :
# version_path_separator = ;
# version_path_separator = space
# version_path_separator = newline
```

```
#
# Use os.pathsep. Default configuration used for new projects.
version_path_separator = os

# set to 'true' to search source files recursively
# in each "version_locations" directory
# new in Alembic version 1.10
# recursive_version_locations = false

# the output encoding used when revision files
# are written from script.py.mako
# output_encoding = utf-8

sqlalchemy.url = "mysql+aiomysql://root:Malato#01@127.0.0.1:3306/user_managem
ent" # driver://user:pass@localhost/dbname

[post_write_hooks]
# post_write_hooks defines scripts or Python functions that are run
# on newly generated revision scripts.  See the documentation for further
# detail and examples

# format using "black" - use the console_scripts runner, against the "black"
entrypoint
# hooks = black
# black.type = console_scripts
# black.entrypoint = black
# black.options = -l 79 REVISION_SCRIPT_FILENAME

# lint with attempts to fix using "ruff" - use the exec runner, execute a bin
ary
# hooks = ruff
# ruff.type = exec
# ruff.executable = %(here)s/.venv/bin/ruff
# ruff.options = check --fix REVISION_SCRIPT_FILENAME

# Logging configuration
[loggers]
keys = root,sqlalchemy,alembic

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = WARNING
handlers = console
qualname =

[logger_sqlalchemy]
level = WARNING
handlers =
qualname = sqlalchemy.engine

[logger_alembic]
level = INFO
handlers =
qualname = alembic

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic
```

```
[formatter_generic]
format = %(levelname)-5.5s [%(name)s] %(message)s
datefmt = %H:%M:%S
```

7. Arquivo: composition_root.py

```
# ./src/dev_platform/infrastructure/composition_root.py
from typing import List, Optional

from application.user.use_cases import (
    CreateUserUseCase,
    ListUsersUseCase,
    UpdateUserUseCase,
    GetUserUseCase,
    DeleteUserUseCase
)
from infrastructure.database.unit_of_work import SQLUnitOfWork
from infrastructure.logging.structured_logger import StructuredLogger
from domain.user.services import (
    UserDomainService,
    UserAnalyticsService,
    DomainServiceFactory
)
from infrastructure.config import CONFIG


class CompositionRoot:
    """
    Composition root for dependency injection.
    Centralizes the creation and configuration of all application dependencie
s.
    """

    def __init__(self):
        # self._config = config or {}
        self._logger = StructuredLogger()
        self._uow = None
        self._domain_service_factory = DomainServiceFactory()

    @property
    def uow(self) -> SQLUnitOfWork:
        if self._uow is None:
            self._uow = SQLUnitOfWork()
        return self._uow

    @property
    def domain_service_factory(self) -> DomainServiceFactory:
        if self._domain_service_factory is None:
            self._domain_service_factory = DomainServiceFactory()
        return self._domain_service_factory

    @property
    def create_user_use_case(self) -> CreateUserUseCase:
        return CreateUserUseCase(
            uow=self.uow,
            user_domain_service=self.domain_service_factory.user_domain_servi
ce,
            logger=self._logger
        )

    @property
    def list_users_use_case(self) -> ListUsersUseCase:
        return ListUsersUseCase(
            uow=self.uow,
            logger=self._logger
        )

    @property
    def update_user_use_case(self) -> UpdateUserUseCase:
        return UpdateUserUseCase(
            uow=self.uow,
            user_domain_service=self.domain_service_factory.user_domain_servi
ce,
            logger=self._logger
        )

    @property
    def get_user_use_case(self) -> GetUserUseCase:
        return GetUserUseCase(
            uow=self.uow,
            logger=self._logger
        )

    @property
    def delete_user_use_case(self) -> DeleteUserUseCase:
        return DeleteUserUseCase(
            uow=self.uow,
            logger=self._logger
        )

    # Domain Services
    def user_domain_service(self, user_repository) -> UserDomainService:
        """
        Create UserDomainService with configuration-based rules.
        """
        # Get configuration for validation rules
        validation_config = CONFIG.get('validation', {})

        return self.domain_service_factory.create_user_domain_service(
            user_repository=user_repository,
            enable_profanity_filter=validation_config.get('enable_profanity_f
ilter', False),
            allowed_domains=validation_config.get('allowed_domains'),
            business_hours_only=validation_config.get('business_hours_only',
False)
        )

    def user_analytics_service(self, user_repository) -> UserAnalyticsService
:
        """Create UserAnalyticsService."""
        return self.domain_service_factory.create_analytics_service(user_repo
sitory)

    # Utility methods for specific configurations
    def create_enterprise_user_domain_service(self, user_repository) -> UserD
omainService:
        """
        Create UserDomainService with enterprise-level validation rules.
        """
        return self.domain_service_factory.create_user_domain_service(
            user_repository=user_repository,
            enable_profanity_filter=True,
            allowed_domains=['empresa.com', 'company.com'],
            business_hours_only=True
        )
```

8. Arquivo: config.py

```
# ./src/dev_platform/infrastructure/config.py
from dotenv import load_dotenv
import os
import json
from typing import Dict, Any, Optional
import warnings
from domain.user.exceptions import ConfigurationException


# Definição de exceções para configuração
class ConfigurationException(Exception):
    pass

class Configuration:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        # A flag para garantir que a inicialização ocorra apenas uma vez por
instância singleton
        if not hasattr(self, '_initialized') or not self._initialized:
            self._initialized = False # Garante que a flag seja redefinida se
 a instância já existia, mas não inicializada
            self._environment = os.getenv("ENVIRONMENT", "production") # Gara
nte que ENVIRONMENT seja lido primeiro
            self._config = {}
            self._load_environment_variables()
            self._load_config_file()
            self._validate_production_config()
            self._initialized = True # Marca como inicializado

    def _load_environment_variables(self):
        """
        Carrega variáveis de ambiente de um arquivo .env específico do ambien
te.
        Por exemplo, se ENVIRONMENT=development, ele tentará carregar .env.de
velopment.
        """
        dotenv_path = f".env.{self._environment}"

        # O base_dir é importante se o script não for executado da raiz do pr
ojeto.
        # Assumindo que os arquivos .env estão na raiz do projeto.
        base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '.
.', '..', '..'))
        full_dotenv_path = os.path.join(base_dir, dotenv_path)

        if os.path.exists(full_dotenv_path):
            load_dotenv(dotenv_path=full_dotenv_path, override=True)
            # print(f"INFO: Carregado .env de {full_dotenv_path}")
        else:
            # Para produção, pode ser normal que as variáveis de ambiente ven
ham do deploy.
            # Para outros ambientes, avise se o arquivo não for encontrado.
            if self._environment == "production":
                print(f"AVISO: Arquivo .env.{self._environment} não encontrad
o em {full_dotenv_path}. Assumindo que as variáveis de ambiente são configura
das externamente para produção.")
            else:
                warnings.warn(f"AVISO: Arquivo .env.{self._environment} não e
ncontrado em {full_dotenv_path}. Algumas variáveis de ambiente podem não esta
r definidas.")

    def _load_config_file(self):
        """
        Carrega e mescla configurações de arquivos JSON específicos do ambien
te.
        Ex: config.development.json, config.test.json.
        """
        config_file_path = f"config.{self._environment}.json"

        base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '.
.', '..', '..'))
        full_config_file_path = os.path.join(base_dir, config_file_path)

        if os.path.exists(full_config_file_path):
            try:
                with open(full_config_file_path, 'r') as f:
                    print(f"INFO: Abrindo arquivo de configuração {full_confi
g_file_path}")
                    environment_config = json.load(f)
                    self._config.update(environment_config)
                    print(f"INFO: Carregado arquivo de configuração {full_con
fig_file_path}")
            except Exception as e:
                warnings.warn(f"Erro ao carregar o arquivo de configuração {f
ull_config_file_path}: {e}")
        else:
            # print(f"INFO: Arquivo de configuração {full_config_file_path} n
ão encontrado. Usando apenas variáveis de ambiente e padrões.")
            pass

    def _validate_production_config(self):
        """Valida que a DATABASE_URL esteja presente em ambiente de produção.
"""
        if self._environment == "production":
            # Agora verifica diretamente de os.getenv, que já foi populado pe
lo load_dotenv
            if not os.getenv("DATABASE_URL"):
                raise ConfigurationException("DATABASE_URL must be set in pro
duction environment.")

    def get(self, key: str, default: Any = None) -> Any:
        """
        Obtém um valor de configuração, preferindo variáveis de ambiente.
        Converte a chave de ponto (ex: 'logging.level') para underscore maiús
culo (ex: 'LOGGING_LEVEL').
        """
        env_key = key.upper().replace('.', '_')
        env_value = os.getenv(env_key)
        if env_value is not None:
            return env_value
        # Se não estiver nas variáveis de ambiente, tenta pegar do arquivo JS
ON (se carregado)
        return self._config.get(key, default)

    def get_all_config(self) -> Dict[str, Any]:
        """Retorna todas as configurações carregadas (mescladas de arquivos e
 ambiente)."""
        # Itera sobre os atributos que se parecem com chaves de configuração
e os combina com _config
        # ou, mais simples, crie um dicionário combinando as variáveis de amb
iente com as configs de arquivo
        all_configs = self._config.copy()
        # Adiciona variáveis de ambiente que podem não estar no _config
        for env_key, env_value in os.environ.items():
            # Pode-se adicionar uma lógica para filtrar apenas variáveis rele
vantes se necessário
            all_configs[env_key.lower().replace('_', '.')] = env_value
        return all_configs

    def _ensure_async_driver(self, url: str) -> str:
        """Garante que a URL do banco de dados use um driver assíncrono."""
        if url.startswith("mysql://"):
            return url.replace("mysql://", "mysql+aiomysql://")
        elif url.startswith("postgresql://"):
            return url.replace("postgresql://", "postgresql+asyncpg://")
        elif url.startswith("sqlite:///"):
            return url.replace("sqlite:///", "sqlite+aiosqlite:///")
        return url

    @property
    def database_url(self) -> str:
        """Retorna a URL do banco de dados com driver assíncrono garantido.""
"
        url = self.get("DATABASE_URL")
        if not url:
            raise ConfigurationException("DATABASE_URL is not configured for
the current environment.")
        return self._ensure_async_driver(url)

    @property
    def sync_database_url(self) -> str:
        """Retorna a URL do banco de dados sem garantir driver assíncrono (pa
ra ferramentas síncronas)."""
        url = self.get("DATABASE_URL")
        if not url:
            raise ConfigurationException("DATABASE_URL is not configured for
the current environment.")
        return url

# Instância singleton da configuração
CONFIG = Configuration()
```

9. Arquivo: dtos.py

```
# ./src/dev_platform/application/user/dtos.py
from pydantic import BaseModel, validator, validate_arguments, StrictStr, Val
idationError, ValidationInfo, field_validator


class UserDTO(BaseModel):
    id: str
    name: StrictStr
    email: StrictStr

    @classmethod
    def from_entity(cls, entity):
        return cls(
            id=str(entity.id),
            name=entity.name.value,
            email=entity.email.value
        )

    def to_entity(self):
        from domain.user.entities import User  # Importar aqui para evitar de
pendência circular
        return User.create(name=self.name, email=self.email)

class UserCreateDTO(BaseModel):
    name: StrictStr
    email: StrictStr

    @field_validator('name')
    def validate_name(cls, v):
        if not v or len(v) == 0:
            raise ValueError("Precisar ser um nome, o campo não pode ficar va
zio")
        return v.strip()

    @field_validator('email')
    def validate_email(cls, v):
        # Validação básica antes de criar Value Object
        if not v or len(v) == 0:
            raise ValueError("Precisar ser um e-mail")
        return v.lower().strip()

class UserUpdateDTO(BaseModel):
    name: StrictStr
    email: StrictStr

    @field_validator('name')
    def validate_name(cls, v):
        return v.strip()

    @field_validator('email')
    def validate_email(cls, v):
        return v.lower().strip()
```

10. Arquivo: entities.py

```
# ./src/dev_platform/domain/user/entities.py
from dataclasses import dataclass
from typing import Optional
from domain.user.value_objects import Email, UserName


@dataclass(frozen=True)
class User:
    id: Optional[int]
    name: UserName
    email: Email

    @classmethod
    def create(cls, name: str, email: str) -> 'User':
        return cls(
            id=None,
            name=UserName(name),
            email=Email(email)
        )

    def with_id(self, new_id: int) -> 'User':
        return User(new_id, self.name, self.email)
```

11. Arquivo: env.py

```
# ./migrations/env.py
from logging.config import fileConfig

from sqlalchemy import engine_from_config
from sqlalchemy import pool
from alembic import context

# Isso importa a instância singleton da sua configuração
# Assumindo que o caminho é src.dev_platform.infrastructure.config
# Ajuste o import path conforme a estrutura real do seu projeto.
import os
import sys

# Adiciona o diretório raiz do projeto ao sys.path para imports absolutos fun
cionarem
# Se alembic.ini e .env estiverem na raiz do projeto, e src/dev_platform for
o módulo,
# precisamos garantir que o sys.path inclua o diretório que contém 'src'.
# O Alembic geralmente é executado da raiz do projeto, então isso é crucial.
# Descobre o caminho do alembic.ini e navega para a raiz do projeto.
current_dir = os.path.dirname(os.path.abspath(__file__))
project_root = os.path.abspath(os.path.join(current_dir, '..', '..')) # Ajust
e conforme a profundidade de 'migrations'
sys.path.insert(0, project_root)

# Agora podemos importar a configuração do seu projeto
# Ajuste o caminho se a sua `config.py` não estiver diretamente em infrastruc
ture
from src.dev_platform.infrastructure.config import CONFIG, ConfigurationExcep
tion

# This is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

# Interpret the config file for Python logging.
# This line sets up loggers basically.
if config.config_file_name is not None:
    fileConfig(config.config_file_name)

# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import Base
# target_metadata = Base.metadata
# Exemplo: Se você tem um models.py na camada de infraestrutura
from src.dev_platform.infrastructure.database.models import Base # Ajuste est
e import para seu modelo base
target_metadata = Base.metadata

# other values from the config, defined by the needs of env.py,
# can be acquired:
# my_important_option = config.get_main_option("my_important_option")
# ... etc.


# --- Começo da Modificação para usar sua CONFIG ---
def get_database_url_from_project_config():
    try:
        # Acesse a URL do banco de dados diretamente da sua instância CONFIG
        # Isso garante que a URL seja carregada dos arquivos .env corretament
e.
        return CONFIG.sync_database_url # Use sync_database_url para Alembic,
 pois ele é síncrono.
```

```
    except ConfigurationException as e:
        print(f"ERRO: Não foi possível obter DATABASE_URL da configuração do
projeto: {e}")
        # Isso é um erro fatal para o Alembic, então re-lançar ou sair.
        sys.exit(1)

# Pega a URL do banco de dados da sua configuração customizada
database_url = get_database_url_from_project_config()

# Define a URL do banco de dados para o contexto do Alembic
config.set_main_option("sqlalchemy.url", database_url)

# --- Fim da Modificação ---

def run_migrations_offline() -> None:
    """Run migrations in 'offline' mode.

    This configures the context with just a URL
    and not an Engine, though an Engine is additionally
    permissible.  By not creating an Engine, we don't even
    need a DBAPI to be available.

    Calls to context.execute() here emit the given string to the
    script output.

    """
    url = config.get_main_option("sqlalchemy.url")
    context.configure(
        url=url,
        target_metadata=target_metadata,
        literal_binds=True,
        dialect_opts={"paramstyle": "named"},
    )

    with context.begin_transaction():
        context.run_migrations()

def run_migrations_online() -> None:
    """Run migrations in 'online' mode.

    In this scenario we need to create an Engine
    and associate a connection with the context.

    """
    # config.url já está definido pelo get_database_url_from_project_config()
 acima
    connectable = engine_from_config(
        CONFIG.get_section_arg(config.config_ini_section),
        prefix="sqlalchemy.",
        poolclass=pool.NullPool,
    )

    with connectable.connect() as connection:
        context.configure(
            connection=connection, target_metadata=target_metadata
        )

        with context.begin_transaction():
            context.run_migrations()

if context.is_offline_mode():
    run_migrations_offline()
else:
    run_migrations_online()
```

12. Arquivo: exceptions.py

```
# ./src/dev_platform/domain/user/exceptions.py
from datetime import datetime
from typing import Optional, Dict, Any


# Application layer exceptions
class ApplicationException(Exception):
    """Base exception for application layer errors."""

    def __init__(self, message: str, original_exception: Optional[Exception]
= None):
        self.message = message
        self.original_exception = original_exception
        self.timestamp = datetime.now()
        super().__init__(self.message)

class UseCaseException(ApplicationException):
    """Raised when a use case execution fails."""

    def __init__(self, use_case_name: str, reason: str, original_exception: O
ptional[Exception] = None):
        self.use_case_name = use_case_name
        self.reason = reason
        super().__init__(
            message=f"Use case '{use_case_name}' failed: {reason}",
            original_exception=original_exception
        )

# Infrastructure layer exceptions
class InfrastructureException(Exception):
    """Base exception for infrastructure layer errors."""

    def __init__(self, message: str, component: str, original_exception: Opti
onal[Exception] = None):
        self.message = message
        self.component = component
        self.original_exception = original_exception
        self.timestamp = datetime.now()
        super().__init__(self.message)

    def to_dict(self) -> Dict[str, Any]:
        """Convert exception to dictionary for logging/serialization."""
        return {
            "message": self.message,
            "component": self.component,
            "timestamp": self.timestamp.isoformat(),
            "original_error": str(self.original_exception) if self.original_e
xception else None
        }

class DatabaseException(InfrastructureException):
    """Raised when database operations fail."""

    def __init__(self, operation: str, reason: str, original_exception: Optio
nal[Exception] = None):
        self.operation = operation
        self.reason = reason
        super().__init__(
            message=f"Database operation '{operation}' failed: {reason}",
            component="database",
            original_exception=original_exception
        )
    def to_dict(self) -> Dict[str, Any]:
        """Extended dictionary representation for database errors."""
        base_dict = super().to_dict()
        base_dict.update({
            "operation": self.operation,
            "reason": self.reason
        })
        return base_dict

class ConfigurationException(InfrastructureException):
    """Raised when configuration is invalid or missing."""

    def __init__(self, config_key: str, reason: str):
        self.config_key = config_key
        self.reason = reason
        super().__init__(
            message=f"Configuration error for '{config_key}': {reason}",
            component="configuration"
        )

class CacheException(InfrastructureException):
    """Raised when cache operations fail."""

    def __init__(self, operation: str, key: str, reason: str, original_except
ion: Optional[Exception] = None):
        self.operation = operation
        self.key = key
        self.reason = reason
        super().__init__(
            message=f"Cache {operation} failed for key '{key}': {reason}",
            component="cache",
            original_exception=original_exception
        )

# Repository-specific exceptions
class RepositoryException(InfrastructureException):
    """Base exception for repository layer errors."""

    def __init__(self, repository_name: str, operation: str, reason: str, ori
ginal_exception: Optional[Exception] = None):
        self.repository_name = repository_name
        self.operation = operation
        self.reason = reason
        super().__init__(
            message=f"Repository '{repository_name}' {operation} failed: {rea
son}",
            component="repository",
            original_exception=original_exception
        )

class DataIntegrityException(RepositoryException):
    """Raised when data integrity constraints are violated."""

    def __init__(self, constraint_name: str, details: str, original_exception
: Optional[Exception] = None):
        self.constraint_name = constraint_name
        self.details = details
        super().__init__(
            repository_name="database",
            operation="constraint_validation",
            reason=f"Constraint '{constraint_name}' violated: {details}",
            original_exception=original_exception
        )

class DataCorruptionException(RepositoryException):
    """Raised when data corruption is detected."""
    def __init__(self, entity_type: str, entity_id: str, corruption_details: 
str):
        self.entity_type = entity_type
        self.entity_id = entity_id
        self.corruption_details = corruption_details
        super().__init__(
            repository_name="database",
            operation="data_validation",
            reason=f"{entity_type} {entity_id} has corrupted data: {corruptio
n_details}"
        )

# Exceções Específicas do Domínio
class DomainException(Exception):
    """Base exception for all domain-related errors."""

    def __init__(self, message: str, error_code: str = None, details: Optiona
l[Dict[str, Any]] = None):
        self.message = message
        self.error_code = error_code or self.__class__.__name__
        self.details = details or {}
        self.timestamp = datetime.now()
        super().__init__(self.message)

    def to_dict(self) -> Dict[str, Any]:
        return {
            "error_code": self.error_code,
            "message": self.message,
            "details": self.details,
            "timestamp": self.timestamp.isoformat()
        }

class UserAlreadyExistsException(DomainException):
    """Raised when trying to create a user with an email that already exists.
"""

    def __init__(self, email: str):
        self.email = email
        super().__init__(
            message=f"User with email '{email}' already exists",
            error_code="USER_ALREADY_EXISTS",
            details={"email": email}
        )

class UserNotFoundException(DomainException):
    """Raised when a user cannot be found."""

    def __init__(self, identifier: str, identifier_type: str = "id"):
        self.identifier = identifier
        self.identifier_type = identifier_type
        super().__init__(
            message=f"User not found with {identifier_type}: {identifier}",
            error_code="USER_NOT_FOUND",
            details={"identifier": identifier, "identifier_type": identifier_
type}
        )

class InvalidUserDataException(DomainException):
    """Raised when user data fails validation."""

    def __init__(self, field: str, value: Any, reason: str):
        self.field = field
        self.value = value
        self.reason = reason
        super().__init__(
            message=f"Invalid {field}: {reason}",
            error_code="INVALID_USER_DATA",
            details={"field": field, "value": str(value), "reason": reason}
        )

class UserValidationException(DomainException):
    """Raised when user business rules validation fails."""

    def __init__(self, validation_errors: Dict[str, str]):
        self.validation_errors = validation_errors
        errors_summary = ", ".join([f"{field}: {error}" for field, error in v
alidation_errors.items()])
        super().__init__(
            message=f"User validation failed: {errors_summary}",
            error_code="USER_VALIDATION_FAILED",
            details={"validation_errors": validation_errors}
        )

class EmailDomainNotAllowedException(DomainException):
    """Raised when email domain is not in allowed list."""

    def __init__(self, email: str, domain: str, allowed_domains: list):
        self.email = email
        self.domain = domain
        self.allowed_domains = allowed_domains
        super().__init__(
            message=f"Email domain '{domain}' is not allowed. Allowed domains
: {', '.join(allowed_domains)}",
            error_code="EMAIL_DOMAIN_NOT_ALLOWED",
            details={
                "email": email,
                "domain": domain,
                "allowed_domains": allowed_domains
            }
        )

class UserOperationException(DomainException):
    """Raised when a user operation fails."""

    def __init__(self, operation: str, user_id: int, reason: str):
        self.operation = operation
        self.user_id = user_id
        self.reason = reason
        super().__init__(
            message=f"Failed to {operation} user {user_id}: {reason}",
            error_code="USER_OPERATION_FAILED",
            details={"operation": operation, "user_id": user_id, "reason": re
ason}
        )

# Compatibility aliases (deprecated, use specific exceptions above)
class DomainError(DomainException):
    """Exception for domain-related errors. DEPRECATED: Use DomainException i
nstead."""

    def __init__(self, message: str):
        import warnings
        warnings.warn(
            "DomainError is deprecated. Use DomainException instead.",
            DeprecationWarning,
            stacklevel=2
        )
        super().__init__(message)

class ValidationException(DomainException):
    """Exception for validation-related errors. DEPRECATED: Use UserValidatio
```

```
nException instead."""

    def __init__(self, message: str):
        import warnings
        warnings.warn(
            "ValidationException is deprecated. Use UserValidationException i
nstead.",
            DeprecationWarning,
            stacklevel=2
        )
        super().__init__(message)
```

13. Arquivo: main.py

```
# ./src/dev_platform/main.py
import click
import os
import sys

# Adiciona o diretório raiz do projeto ao sys.path para imports absolutos
current_dir = os.path.dirname(os.path.abspath(__file__))
project_root = os.path.abspath(os.path.join(current_dir, '..', '..'))
if project_root not in sys.path:
    sys.path.insert(0, project_root)

# Importe user_cli do user_commands (renomeado para evitar conflito)
from src.dev_platform.interface.cli.user_commands import cli as user_cli


# Cria um grupo Click principal
@click.group()
def main_cli():
    """CLI para o DEV Platform."""
    pass

# Adiciona os comandos de usuário como um subgrupo 'user'
main_cli.add_command(user_cli, name="user")


if __name__ == '__main__':
    main_cli()
```

14. Arquivo: models.py

```
# ./src/dev_platform/infrastructure/database/models.py
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import declarative_base

Base = declarative_base()


class UserModel(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(100), nullable=False)
    email = Column(String(100), nullable=False, unique=True)
```

15. Arquivo: ports.py

```python
# ./src/dev_platform/application/user/ports.py
from abc import ABC, abstractmethod
from typing import List, Optional
from domain.user.entities import User


class UserRepository(ABC):
    @abstractmethod
    async def save(self, user: User) -> User:
        pass

    @abstractmethod
    async def find_by_email(self, email: str) -> Optional[User]:
        pass

    @abstractmethod
    async def find_all(self) -> List[User]:
        pass

    @abstractmethod
    async def find_by_id(self, user_id: int) -> Optional[User]:
        pass

    @abstractmethod
    async def delete(self, user_id: int) -> bool:
        pass

    @abstractmethod
    async def find_by_name_contains(self, name_part: str) -> List[User]:
        pass

    @abstractmethod
    async def count(self) -> int:
        pass

class Logger(ABC):
    @abstractmethod
    def info(self, message: str, **kwargs):
        pass

    @abstractmethod
    def error(self, message: str, **kwargs):
        pass

    @abstractmethod
    def warning(self, message: str, **kwargs):
        pass

class UnitOfWork(ABC):
    users: UserRepository

    @abstractmethod
    async def __aenter__(self):
        pass

    @abstractmethod
    async def __aexit__(self, exc_type, exc_val, exc_tb):
        pass

    @abstractmethod
    async def commit(self):
        pass
```

16. Arquivo: pyproject.toml

```toml
# ./pyproject.toml
[project]
name = "dev-platform"
version = "0.1.0"
description = ""
authors = [
    {name = "Sergio Pereira",email = "sergiopereira.br@hotmail.com"}
]
readme = "README.md"
requires-python = ">=3.11,<4.0"
dependencies = [
    "pymysql (>=1.1.1,<2.0.0)",
    "sqlalchemy[asyncio] (>=2.0.41,<3.0.0)",
    "aiomysql (>=0.2.0,<0.3.0)",
    "typer (>=0.15.4,<0.16.0)",
    "pydantic (>=2.11.5,<3.0.0)",
    "loguru (>=0.7.3,<0.8.0)",
    "uuid (>=1.30,<2.0)"
]

[tool.poetry]
packages = [{include = "dev_platform", from = "src"}]

[tool.poetry.group.dev.dependencies]
black = "^23.0"
flake8 = "^6.0"
alembic = "^1.15.2"
sqlalchemy = "^2.0.41"
taskipy = "^1.14.1"
reportlab = "^4.4.1"
chardet = "^5.2.0"
pypandoc = "^1.15"
mypy = "^1.16.0"

[tool.poetry.group.docs.dependencies]
mkdocs = "^1.6.1"
pymdown-extensions = "^10.15"
mkdocstrings = "^0.29.1"
mkdocstrings-python = "^1.16.10"
sphinx = "^6.0"
sphinx-rtd-theme = "^1.0"
mkdocs-material = "^9.6.14"


[tool.poetry.group.test.dependencies]
pytest = "^8.3.5"

[build-system]
requires = ["poetry-core>=2.0.0,<3.0.0"]
build-backend = "poetry.core.masonry.api"

# Configuração opcional para definir o script principal
[tool.poetry.scripts]
dev-platform = "dev_platform.main:main"  # Isso cria um comando executável
docs-serve = "mkdocs:serve"  # Iniciar servidor de documentação local
docs-build = "mkdocs:build"  # Construir a documentação estática

[tool.taskipy.tasks]
list = "poetry run python ./src/dev_platform/main.py list-users"
clean = '.\\scripts\\tools\\del_folderes.ps1 ".\\src" "__pycache__"'
```

17. Arquivo: repositories.py

```
# ./src/dev_platform/infrastructure/database/repositories.py
from typing import List, Optional
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select
from sqlalchemy import delete, func
from sqlalchemy.exc import SQLAlchemyError, IntegrityError
from application.user.ports import UserRepository
from domain.user.entities import User
from domain.user.value_objects import UserName, Email
from domain.user.exceptions import (
    DatabaseException,
    UserAlreadyExistsException,
    UserNotFoundException
)
from infrastructure.database.models import UserModel


class SQLUserRepository(UserRepository):
    def __init__(self, session: AsyncSession):
        self._session = session

    def _handle_database_error(self, operation: str, error: Exception, **cont
ext):
        """Centralized error handling for database operations."""
        if isinstance(error, IntegrityError):
            # Check if it's a unique constraint violation
            if "email" in str(error.orig).lower() and "unique" in str(error.o
rig).lower():
                email = context.get('email', 'unknown')
                raise UserAlreadyExistsException(email)

        # Log context information for debugging
        context_str = ", ".join([f"{k}={v}" for k, v in context.items()])
        error_msg = f"{operation} failed"
        if context_str:
            error_msg += f" ({context_str})"

        raise DatabaseException(
            operation=operation,
            reason=str(error),
            original_exception=error
        )

    def _convert_to_domain_user(self, db_user: UserModel) -> User:
        """Convert database model to domain entity."""
        try:
            return User(
                id=db_user.id,
                name=UserName(db_user.name),
                email=Email(db_user.email)
            )
        except ValueError as e:
            # This should not happen if database constraints are properly set
            raise DatabaseException(
                operation="data_conversion",
                reason=f"Invalid data in database: {str(e)}",
                original_exception=e
            )

    async def save(self, user: User) -> User:
        """Save a user to the database."""
        try:
            if user.id is None:
                # Create new user
                db_user = UserModel(
                    name=user.name.value,
                    email=user.email.value
                )
                self._session.add(db_user)
                await self._session.flush()

                # Return user with the generated ID
                return User(
                    id=db_user.id,
                    name=user.name,
                    email=user.email
                )
            else:
                # Update existing user
                result = await self._session.execute(
                    select(UserModel).where(UserModel.id == user.id)
                )
                db_user = result.scalars().first()

                if not db_user:
                    raise UserNotFoundException(str(user.id))

                db_user.name = user.name.value
                db_user.email = user.email.value
                await self._session.flush()

                return User(
                    id=db_user.id,
                    name=user.name,
                    email=user.email
                )

        except UserNotFoundException:
            # Re-raise domain exceptions as-is
            raise
        except UserAlreadyExistsException:
            # Re-raise domain exceptions as-is
            raise
        except SQLAlchemyError as e:
            self._handle_database_error(
                operation="save_user",
                error=e,
                user_id=user.id,
                email=user.email.value
            )
        except Exception as e:
            self._handle_database_error(
                operation="save_user",
                error=e,
                user_id=user.id,
                email=user.email.value
            )

    async def find_by_email(self, email: str) -> Optional[User]:
        """Find a user by email address."""
        try:
            result = await self._session.execute(
                select(UserModel).where(UserModel.email == email)
            )
            db_user = result.scalars().first()

            if db_user:
                return self._convert_to_domain_user(db_user)
            return None
        except SQLAlchemyError as e:
            self._handle_database_error(
                operation="find_by_email",
                error=e,
                email=email
            )
        except Exception as e:
            self._handle_database_error(
                operation="find_by_email",
                error=e,
                email=email
            )
        return None

    async def find_all(self) -> List[User]:
        """Find all users in the database."""
        try:
            result = await self._session.execute(select(UserModel))
            db_users = result.scalars().all()

            return [
                self._convert_to_domain_user(db_user)
                for db_user in db_users
            ]

        except SQLAlchemyError as e:
            self._handle_database_error(
                operation="find_all_users",
                error=e
            )
        except Exception as e:
            self._handle_database_error(
                operation="find_all_users",
                error=e
            )

        return [] # Nota de teste: Retornar None aqui pode ser problemático,
pois o método deve retornar uma lista vazia se não houver usuários. Considere
 retornar uma lista vazia em vez de None.

    async def find_by_id(self, user_id: int) -> Optional[User]:
        """Find a user by ID."""
        try:
            result = await self._session.execute(
                select(UserModel).where(UserModel.id == user_id)
            )
            db_user = result.scalars().first()

            if db_user:
                return self._convert_to_domain_user(db_user)
            return None

        except SQLAlchemyError as e:
            self._handle_database_error(
                operation="find_by_id",
                error=e,
                user_id=user_id
            )
        except Exception as e:
            self._handle_database_error(
                operation="find_by_id",
                error=e,
                user_id=user_id
            )
        return None

    async def find_by_ids(self, user_ids: List[int]) -> List[User]:
        result = await self._session.execute(
            select(UserModel).where(UserModel.id.in_(user_ids))
        )
        if result is None:
            return [] # Nota de teste: Verificar resultado

        return [self._convert_to_domain_user(u) for u in result.scalars().all
()]

    async def delete(self, user_id: int) -> bool:
        """Delete a user by ID."""
        try:
            # First check if user exists
            existing_user = await self.find_by_id(user_id)
            if not existing_user:
                raise UserNotFoundException(str(user_id))

            # Perform deletion
            result = await self._session.execute(
                delete(UserModel).where(UserModel.id == user_id)
            )

            success = result.rowcount > 0
            if success:
                await self._session.flush()

            return success

        except UserNotFoundException:
            # Re-raise domain exceptions as-is
            raise
        except SQLAlchemyError as e:
            self._handle_database_error(
                operation="delete_user",
                error=e,
                user_id=user_id
            )
        except Exception as e:
            self._handle_database_error(
                operation="delete_user",
                error=e,
                user_id=user_id
            ) # Nota de teste: Retornar False se a exclusão falhar, o que é m
ais intuitivo do que retornar None.
        return False

    async def find_by_name_contains(self, name_part: str) -> List[User]:
        """Find users whose name contains the given string."""
        try:
            result = await self._session.execute(
                select(UserModel).where(UserModel.name.contains(name_part))
            )
            db_users = result.scalars().all()

            return [
                self._convert_to_domain_user(db_user)
                for db_user in db_users
            ]

        except SQLAlchemyError as e:
            self._handle_database_error(
                operation="find_by_name_contains",
                error=e,
                name_part=name_part
            )
        except Exception as e:
            self._handle_database_error(
                operation="find_by_name_contains",
                error=e,
                name_part=name_part
            )

        return []

    async def count(self) -> int: # Nota
        """Count total number of users."""
        try:
            result = await self._session.execute(
                select(func.count(UserModel.id))
            )
            count = result.scalar()
            return count if count is not None else 0

        except SQLAlchemyError as e:
            self._handle_database_error(
                operation="count_users",
                error=e
            )
        except Exception as e:
            self._handle_database_error(
                operation="count_users",
                error=e
            )

        return 0 # Nota de teste: Retornar 0 se a contagem falhar, o que é ma
is intuitivo do que retornar None.

class RepositoryExceptionHandler:
    """Utility class for handling repository exceptions consistently."""

    @staticmethod
    def handle_sqlalchemy_error(operation: str, error: SQLAlchemyError, **con
text):
        """Handle SQLAlchemy specific errors."""
        if isinstance(error, IntegrityError):
            if "email" in str(error.orig).lower() and "unique" in str(error.o
rig).lower():
                email = context.get('email', 'unknown')
                raise UserAlreadyExistsException(email)

        context_str = ", ".join([f"{k}={v}" for k, v in context.items()])
        error_msg = f"{operation} failed"
        if context_str:
            error_msg += f" ({context_str})"

        raise DatabaseException(
            operation=operation,
            reason=str(error),
            original_exception=error
        )

    @staticmethod
    def handle_generic_error(operation: str, error: Exception, **context):
        """Handle generic errors."""
        context_str = ", ".join([f"{k}={v}" for k, v in context.items()])
        error_msg = f"{operation} failed"
        if context_str:
            error_msg += f" ({context_str})"
```

```
raise DatabaseException(
    operation=operation,
    reason=str(error),
    original_exception=error
)
```

18. Arquivo: services.py

```
# ./src/dev_platform/domain/user/services.py
from abc import ABC, abstractmethod
from typing import List, Dict, Optional, Set
import re
from datetime import datetime, timedelta
from domain.user.entities import User
from domain.user.exceptions import (
    UserAlreadyExistsException,
    UserNotFoundException,
    EmailDomainNotAllowedException,
    UserValidationException,
    InvalidUserDataException
)


class UserUniquenessService:
    """Service focused on uniqueness validation."""

    def __init__(self, user_repository):
        self._repository = user_repository

    async def ensure_email_is_unique(self, email: str, exclude_user_id: Optio
nal[int] = None) -> None:
        existing_user = await self._repository.find_by_email(email)
        if existing_user and (exclude_user_id is None or existing_user.id !=
exclude_user_id):
            # from domain.user.exceptions import UserAlreadyExistsException
            raise UserAlreadyExistsException(email)

class ValidationRule(ABC):
    """Base class for validation rules."""

    @abstractmethod
    async def validate(self, user: User) -> Optional[str]:
        """
        Validate user according to this rule.
        Returns None if valid, error message if invalid.
        """
        pass

    @property
    @abstractmethod
    def rule_name(self) -> str:
        pass

class EmailDomainValidationRule(ValidationRule):
    """Validates that email domain is in allowed list."""

    def __init__(self, allowed_domains: List[str]):
        self.allowed_domains = set(domain.lower() for domain in allowed_domai
ns)

    async def validate(self, user: User) -> Optional[str]:
        email_domain = user.email.value.split('@')[1].lower()
        if email_domain not in self.allowed_domains:
            return f"Email domain '{email_domain}' is not allowed. Allowed do
mains: {', '.join(self.allowed_domains)}"
        return None

    @property
    def rule_name(self) -> str:
        return "email_domain_validation"
```

```
class NameProfanityValidationRule(ValidationRule):
    """Validates that name doesn't contain profanity."""

    def __init__(self, forbidden_words: List[str]):
        self.forbidden_words = [word.lower() for word in forbidden_words]

    async def validate(self, user: User) -> Optional[str]:
        name_lower = user.name.value.lower()
        for word in self.forbidden_words:
            if word in name_lower:
                return f"Name contains forbidden word: {word}"
        return None

    @property
    def rule_name(self) -> str:
        return "name_profanity_validation"

class EmailFormatAdvancedValidationRule(ValidationRule):
    """Advanced email format validation beyond basic regex."""

    def __init__(self):
        # More restrictive email validation
        self.pattern = re.compile(
            r'^[a-zA-Z0-9]([a-zA-Z0-9._-]*[a-zA-Z0-9])?@[a-zA-Z0-9]([a-zA-Z0-
9.-]*[a-zA-Z0-9])?\.[a-zA-Z]{2,}$'
        )

    async def validate(self, user: User) -> Optional[str]:
        email = user.email.value

        # Check basic format
        if not self.pattern.match(email):
            return "Email format is invalid"

        # Check for consecutive dots
        if '..' in email:
            return "Email cannot contain consecutive dots"

        # Check for valid length
        if len(email) > 254:
            return "Email is too long (max 254 characters)"

        local_part, domain_part = email.split('@')

        # Check local part length
        if len(local_part) > 64:
            return "Email local part is too long (max 64 characters)"

        # Check domain part
        if len(domain_part) > 253:
            return "Email domain part is too long (max 253 characters)"

        return None

    @property
    def rule_name(self) -> str:
        return "email_format_advanced_validation"

class NameContentValidationRule(ValidationRule):
    """Validates name content and format."""

    def __init__(self, allowed_chars: Optional[Set[str]] = None):
        if allowed_chars is None:
            allowed_chars = set("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
STUVWXYZ -'àáâãèéêìíîòóôõùúûçÀÁÂÃÈÉÊÌÍÎÒÓÔÕÙÚÛÇ")  # Carregue de config exter
na
```

```
        self.allowed_chars = allowed_chars

    async def validate(self, user: User) -> Optional[str]:
        name = user.name.value

        # Check for only whitespace
        if name.strip() != name:
            return "Name cannot start or end with whitespace"

        # Check for excessive whitespace
        if '  ' in name:
            return "Name cannot contain consecutive spaces"

        # Check for numbers
        if any(char.isdigit() for char in name):
            return "Name cannot contain numbers"

        # Check for special characters (allow only letters, spaces, hyphens,
apostrophes)
        if not all(char in self.allowed_chars for char in name):
            invalid_chars = [char for char in name if char not in self.allowe
d_chars]
            return f"Name contains invalid characters: {', '.join(set(invalid
_chars))}"

        # Check minimum word count
        words = name.split()
        if len(words) < 2:
            return "Name must contain at least first and last name"

        # Check each word length
        for word in words:
            if len(word) < 2:
                return "Each name part must be at least 2 characters long"

        return None

    @property
    def rule_name(self) -> str:
        return "name_content_validation"

class BusinessHoursValidationRule(ValidationRule):
    """Example rule that validates based on business hours."""

    def __init__(self, business_hours_only: bool = False):
        self.business_hours_only = business_hours_only

    async def validate(self, user: User) -> Optional[str]:
        if not self.business_hours_only:
            return None

        now = datetime.now()
        # Check if it's business hours (9 AM to 5 PM, Monday to Friday)
        if now.weekday() >= 5:  # Saturday or Sunday
            return "User registration only allowed during business days"

        if now.hour < 9 or now.hour >= 17:
            return "User registration only allowed during business hours (9 A
M - 5 PM)"

        return None

    @property
    def rule_name(self) -> str:
        return "business_hours_validation"
```

```
class UserDomainService:
    """Service for complex user domain validations and business rules."""

    def __init__(self, user_repository, validation_rules: Optional[List] = No
ne):
        self._repository = user_repository
        self._validation_rules = validation_rules or []
        self._setup_default_rules()

    def _setup_default_rules(self):
        """Setup default validation rules if none provided."""
        if not self._validation_rules:
            self._validation_rules = [
                EmailFormatAdvancedValidationRule(),
                NameContentValidationRule(),
                # Add more default rules as needed
            ]

    def add_validation_rule(self, rule: ValidationRule):
        """Add a custom validation rule."""
        self._validation_rules.append(rule)

    def remove_validation_rule(self, rule_name: str):
        """Remove a validation rule by name."""
        self._validation_rules = [
            rule for rule in self._validation_rules
            if rule.rule_name != rule_name
        ]

    async def validate_business_rules(self, user: User) -> None:
        """
        Validate all business rules for a user.
        Raises UserValidationException if any rule fails.
        """
        validation_errors = {}

        # Check uniqueness first
        try:
            await self._validate_unique_email(user.email.value)
        except UserAlreadyExistsException as e:
            validation_errors["email"] = e.message

        # Run all validation rules
        for rule in self._validation_rules:
            try:
                error_message = await rule.validate(user)
                if error_message:
                    validation_errors[rule.rule_name] = error_message
            except Exception as e:
                validation_errors[rule.rule_name] = f"Validation rule failed:
 {str(e)}"

        # If there are validation errors, raise exception
        if validation_errors:
            raise UserValidationException(validation_errors)

    async def _validate_unique_email(self, email: str):
        """Validate that email is unique in the system."""
        existing_user = await self._repository.find_by_email(email)
        if existing_user:
            raise UserAlreadyExistsException(email)

    async def validate_user_update(self, user_id: int, updated_user: User) ->
 None:
        """
        Validate user update, checking uniqueness only if email changed.
        """
        validation_errors = {}

        # Get current user
        current_user = await self._repository.find_by_id(user_id)
        if not current_user:
            raise UserNotFoundException(str(user_id))

        # Check email uniqueness only if email changed
        if current_user.email.value != updated_user.email.value:
            try:
                await self._validate_unique_email(updated_user.email.value)
            except UserAlreadyExistsException as e:
                validation_errors["email"] = e.message

        # Run validation rules
        for rule in self._validation_rules:
            try:
                error_message = await rule.validate(updated_user)
                if error_message:
                    validation_errors[rule.rule_name] = error_message
            except Exception as e:
                validation_errors[rule.rule_name] = f"Validation rule failed:
 {str(e)}"

        if validation_errors:
            raise UserValidationException(validation_errors)

    def get_validation_summary(self) -> Dict[str, str]:
        """Get summary of all active validation rules."""
        return {
            rule.rule_name: rule.__class__.__doc__ or "No description availab
le"
            for rule in self._validation_rules
        }

    async def validate_user_creation_constraints(self, user: User) -> None:
        """
        Validate constraints specific to user creation.
        This can include rate limiting, domain restrictions, etc.
        """
        validation_errors = {}

        # Example: Check if we've reached user limit for the day
        # This is just an example - you'd implement based on your business ru
les
        try:
            current_count = await self._repository.count()
            if current_count >= 10000:  # Example limit
                validation_errors["system_limit"] = "Maximum number of users
reached"
        except Exception as e:
            validation_errors["system_check"] = f"Unable to verify system con
straints: {str(e)}"

        if validation_errors:
            raise UserValidationException(validation_errors)

    async def validate_business_domain_rules(self, user: User, domain_whiteli
st: Optional[List[str]] = None) -> None:
        """
        Validate business-specific domain rules.
        """
        if domain_whitelist:
            email_domain = user.email.value.split('@')[1].lower()
            if email_domain not in [d.lower() for d in domain_whitelist]:
                raise EmailDomainNotAllowedException(
                    user.email.value,
                    email_domain,
                    domain_whitelist
                )

class UserAnalyticsService:
    """Service for user analytics and reporting."""

    def __init__(self, user_repository):
        self._repository = user_repository

    async def get_user_statistics(self) -> Dict[str, int]:
        """Get basic user statistics."""
        try:
            total_users = await self._repository.count()

            # You could add more analytics here
            return {
                "total_users": total_users,
                # Add more metrics as needed
            }
        except Exception as e:
            raise RuntimeError(f"Failed to get user statistics: {str(e)}")

    async def find_users_by_domain(self, domain: str) -> List[User]:
        """Find all users with emails from a specific domain."""
        try:
            all_users = await self._repository.find_all()
            return [
                user for user in all_users
                if user.email.value.split('@')[1].lower() == domain.lower()
            ]
        except Exception as e:
            raise RuntimeError(f"Failed to find users by domain: {str(e)}")

# Factory for creating domain services with common configurations
class DomainServiceFactory:
    """Factory for creating domain services with common configurations."""

    def __init__(self):
        pass  # Permite instanciação

    def create_user_domain_service(
        self,
        user_repository,
        enable_profanity_filter: bool = False,
        allowed_domains: Optional[List[str]] = None,
        business_hours_only: bool = False
    ) -> UserDomainService:
        """Create a UserDomainService with common rule configurations."""

        rules = ['',
            EmailFormatAdvancedValidationRule(),
            NameContentValidationRule()
        ]

        if enable_profanity_filter:
            # Add common profanity words - in production, load from config/da
tabase
            forbidden_words = ["badword1", "badword2"]  # Replace with actual
 list
            rules.append(NameProfanityValidationRule(forbidden_words))

        if allowed_domains:
            rules.append(EmailDomainValidationRule(allowed_domains))
        if business_hours_only:
            rules.append(BusinessHoursValidationRule(business_hours_only))

        return UserDomainService(user_repository, rules)

    def create_analytics_service(self, user_repository) -> UserAnalyticsServi
ce:
        """Create a UserAnalyticsService."""
        return UserAnalyticsService(user_repository)
```

19. Arquivo: session.py

```
# ./src/dev_platform/infrastructure/database/session.py
from contextlib import asynccontextmanager
from typing import AsyncGenerator
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession, async_s
essionmaker
from infrastructure.config import CONFIG


class DatabaseSessionManager:
    """Gerenciador centralizado de sessões de banco de dados."""

    def __init__(self):
        self._async_engine = None
        self._sync_engine = None
        self._async_session_factory = None
        self._sync_session_factory = None
        self._initialize_engines()

    def _initialize_engines(self):
        """Inicializa os engines síncronos e assíncronos."""
        # Configurações do pool
        pool_config = {
            "pool_size": CONFIG.get("database.pool_size", 5),
            "max_overflow": CONFIG.get("database.max_overflow", 10),
            "pool_pre_ping": CONFIG.get("database.pool_pre_ping", True)
        }

        # Engine assíncrono
        async_url = CONFIG.get("DATABASE_URL")
        self._async_engine = create_async_engine(
            async_url,
            echo=CONFIG.get("database.echo", False),
            **pool_config
        )

        # Session factory assíncrona
        self._async_session_factory = async_sessionmaker(
            bind=self._async_engine,
            class_=AsyncSession,
            expire_on_commit=False
        )

        # Engine síncrono (se necessário para migrações ou outras operações)
        if not async_url.startswith("sqlite+aiosqlite"):  # SQLite não precis
a de engine síncrono separado
            sync_url = CONFIG.get("DATABASE_URL")
            self._sync_engine = create_engine(
                sync_url,
                echo=CONFIG.get("database.echo", False),
                **pool_config
            )

            self._sync_session_factory = sessionmaker(
                bind=self._sync_engine,
                autocommit=False,
                autoflush=False
            )

    @asynccontextmanager
    async def get_async_session(self) -> AsyncGenerator[AsyncSession, None]:
        """Context manager para sessões assíncronas de banco de dados."""
        async with self._async_session_factory() as session:
            try:
                yield session
                await session.commit()
            except Exception:
                await session.rollback()
                raise

    def get_sync_session(self):
        """Obtém uma sessão síncrona (para migrações, etc.)."""
        if not self._sync_session_factory:
            raise RuntimeError("Sync session not available for this database
type")
        return self._sync_session_factory()

    async def close_async_engine(self):
        """Fecha o engine assíncrono."""
        if self._async_engine:
            await self._async_engine.dispose()

    def close_sync_engine(self):
        """Fecha o engine síncrono."""
        if self._sync_engine:
            self._sync_engine.dispose()

    @property
    def async_engine(self):
        """Propriedade para acessar o engine assíncrono."""
        return self._async_engine

    @property
    def sync_engine(self):
        """Propriedade para acessar o engine síncrono."""
        return self._sync_engine


# Instância global do gerenciador de sessões
db_manager = DatabaseSessionManager()

# Funções de conveniência para compatibilidade
async def get_async_session():
    """Função de conveniência para obter sessão assíncrona."""
    async with db_manager.get_async_session() as session:
        yield session

def get_sync_session():
    """Função de conveniência para obter sessão síncrona."""
    return db_manager.get_sync_session()

# Aliases para compatibilidade com código existente
AsyncSessionLocal = db_manager._async_session_factory
if db_manager._sync_session_factory:
    SessionLocal = db_manager._sync_session_factory
```

20. Arquivo: structured_logger.py

```
# src/dev_platform/infrastructure/logging/structured_logger.py

from typing import Dict, Any, Optional
import os
from uuid import uuid4
from loguru import logger
from infrastructure.config import CONFIG
from application.user.ports import Logger as LoggerPort


class StructuredLogger(LoggerPort):
    """Logger estruturado usando Loguru com suporte a níveis dinâmicos e corr
elação de logs."""

    def __init__(self, name: str = "DEV Platform"):
        self._name = name
        self._configure_logger()

    def _configure_logger(self):
        """Configura o logger com base no ambiente e adiciona handlers."""
        # Remover handlers padrão do Loguru
        logger.remove()

        # Obter nível de log com base no ambiente
        environment = CONFIG.get("environment", "production")
        log_level = CONFIG.get("logging.level", "INFO").upper()
        log_levels = {
            "development": "DEBUG",
            "test": "DEBUG",
            "production": "INFO"
        }
        default_level = log_levels.get(environment, "INFO")
        final_level = log_level if log_level in ["DEBUG", "INFO", "WARNING",
"ERROR", "CRITICAL"] else default_level

        # Configurar handler para console (JSON, todos os níveis)
        logger.add(
            sink="sys.stdout",
            level=final_level,
            format="{time:YYYY-MM-DD HH:mm:ss.SSS} | {level} | {message} | {e
xtra}",
            serialize=True  # Formato JSON
        )

        # Configurar handler para arquivo (apenas ERROR, com rotação)
        if not os.path.exists("logs"):
            os.makedirs("logs")
        logger.add(
            sink=f"logs/{self._name}_{{time:YYYY-MM-DD}}.log",
            level="ERROR",
            rotation="10 MB",
            retention="5 days",
            compression="zip",
            enqueue=True  # Assíncrono
        )

    def set_correlation_id(self, correlation_id: Optional[str] = None):
        """Define um ID de correlação para rastreamento."""
        logger.contextualize(correlation_id=correlation_id or str(uuid4()))

    def info(self, message: str, **kwargs):
        """Registra uma mensagem de nível INFO."""
        logger.bind(**kwargs).info(message)
    def error(self, message: str, **kwargs):
        """Registra uma mensagem de nível ERROR."""
        logger.bind(**kwargs).error(message)

    def warning(self, message: str, **kwargs):
        """Registra uma mensagem de nível WARNING."""
        logger.bind(**kwargs).warning(message)

    def debug(self, message: str, **kwargs):
        """Registra uma mensagem de nível DEBUG."""
        logger.bind(**kwargs).debug(message)

    def critical(self, message: str, **kwargs):
        """Registra uma mensagem de nível CRITICAL."""
        logger.bind(**kwargs).critical(message)

    # NOVO MÉTODO PARA SHUTDOWN GRACIOSO DO LOGGER
    @staticmethod
    def shutdown():
        """
        Garante que todas as mensagens enfileiradas pelo Loguru sejam process
adas
        e que os handlers sejam removidos. Isso é crucial para limpar recurso
s
        assíncronos do logger antes que o loop de eventos feche.
        """
        logger.complete() # Processa todas as mensagens enfileiradas
        logger.remove()  # Remove todos os handlers para evitar vazamentos de
 recursos
```

21. Arquivo: unit_of_work.py

```
# ./src/dev_platform/infrastructure/database/unit_of_work.py
from typing import Optional
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine
from sqlalchemy.orm import sessionmaker

# Importe a CONFIG global
from infrastructure.config import CONFIG
from application.user.ports import UnitOfWork as AbstractUnitOfWork
from infrastructure.database.session import db_manager
from infrastructure.database.repositories import SQLUserRepository


# Estas variáveis devem ser criadas uma única vez na aplicação.
# Poderiam estar em um módulo 'session.py' separado ou aqui,
# mas fora da classe para garantir que não sejam recriadas.
_async_engine = None
_async_session_factory = None

async def get_async_engine():
    """Cria e retorna o engine assíncrono, garantindo que seja um singleton."
""
    global _async_engine
    if _async_engine is None:
        _async_engine = create_async_engine(
            CONFIG.database_url,
            echo=CONFIG.get("DB_ECHO", "False").lower() == "true",
            pool_size=int(CONFIG.get("DB_POOL_SIZE", 10)),
            max_overflow=int(CONFIG.get("DB_MAX_OVERFLOW", 20))
        )
    return _async_engine

async def get_async_session_factory():
    """Cria e retorna a factory de sessão assíncrona, garantindo que seja um
singleton."""
    global _async_session_factory
    if _async_session_factory is None:
        engine = await get_async_engine() # Garante que o engine está criado
        _async_session_factory = sessionmaker(
            engine,
            class_=AsyncSession,
            expire_on_commit=False
        )
    return _async_session_factory

class SQLUnitOfWork(AbstractUnitOfWork):
    def __init__(self):
        self._session: Optional[AsyncSession] = None
        self.users: Optional[SQLUserRepository] = None

    async def __aenter__(self):
        # Usar o gerenciador de sessões
        self._session_context = db_manager.get_async_session()
        self._session = await self._session_context.__aenter__()
        self.users = SQLUserRepository(self._session)
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        if not self._session:
            return

        try:
            if exc_type is None:
                await self._session.commit()
            else:
                await self._session.rollback()
        except Exception as e:
            self._logger.error(f"Error in transaction cleanup: {e}")
            try:
                await self._session.rollback()
            except:
                pass
        finally:
            try:
                await self._session.close()
                await self._session_context.__aexit__(exc_type, exc_val, exc_
tb)
                self._session = None
                self.users = None
            except Exception as e:
                self._logger.error(f"Error closing session: {e}")

    async def commit(self):
        if self._session:
            await self._session.commit()

    async def rollback(self):
        if self._session:
            await self._session.rollback()
```

22. Arquivo: use_cases.py

```
# ./src/dev_platform/application/user/use_cases.py
from typing import List
from application.user.ports import Logger, UnitOfWork
from application.user.dtos import UserCreateDTO
from domain.user.entities import User
from domain.user.services import DomainServiceFactory
from domain.user.exceptions import (
    UserValidationException,
    UserAlreadyExistsException,
    UserNotFoundException,
    DomainException
)


class BaseUseCase:
    def __init__(self, uow: UnitOfWork, logger: Logger):
        self._uow = uow
        self._logger = logger

class CreateUserUseCase(BaseUseCase):
    # CORRIGIDO: Adicionado domain_service_factory como parâmetro
    def __init__(self, uow: UnitOfWork, logger: Logger, domain_service_factor
y: DomainServiceFactory):
        super().__init__(uow, logger)
        self._domain_service_factory = domain_service_factory

    async def execute(self, dto: UserCreateDTO) -> User:
        async with self._uow:
            # Gerar ID de correlação para esta operação
            self._logger.set_correlation_id()

            self._logger.info("Starting user creation", name=dto.name, email=
dto.email)

            try:
                # Create user entity from DTO
                user = User.create(name=dto.name, email=dto.email)

                # Create domain service with repository access
                domain_service = self._domain_service_factory.create_user_dom
ain_service(
                    self._uow.users
                )

                # CORRIGIDO: Método correto é validate_business_rules
                await domain_service.validate_business_rules(user)

                self._logger.info("User validation passed", email=dto.email)

                # Save user
                saved_user = await self._uow.users.save(user)
                await self._uow.commit()

                self._logger.info("User created successfully",
                             user_id=saved_user.id,
                             name=saved_user.name.value,
                             email=saved_user.email.value)

                return saved_user

            except UserValidationException as e:
                self._logger.error("User validation failed",
                              email=dto.email,
                            validation_errors=e.validation_errors)
                raise

            except UserAlreadyExistsException as e:
                self._logger.warning("Attempted to create duplicate user", em
ail=dto.email)
                raise

            except DomainException as e:
                self._logger.error("Domain error during user creation",
                            error_code=e.error_code,
                            message=e.message,
                            details=e.details)
                raise

            except Exception as e:
                self._logger.error("Unexpected error during user creation",
                            email=dto.email,
                            error=str(e))
                raise RuntimeError(f"Failed to create user: {str(e)}")


class ListUsersUseCase(BaseUseCase):
    async def execute(self) -> List[User]:
        async with self._uow:
            try:
                self._logger.info("Starting user listing")
                users = await self._uow.users.find_all()
                self._logger.info("Users retrieved successfully", count=len(u
sers))
                return users

            except Exception as e:
                self._logger.error("Error listing users", error=str(e))
                raise RuntimeError(f"Failed to list users: {str(e)}")


class UpdateUserUseCase(BaseUseCase):
    # CORRIGIDO: Adicionado domain_service_factory como parâmetro
    def __init__(self, uow: UnitOfWork, logger: Logger, domain_service_factor
y: DomainServiceFactory):
        super().__init__(uow, logger)
        self._domain_service_factory = domain_service_factory

    async def execute(self, user_id: int, dto: UserCreateDTO) -> User:
        async with self._uow:
            # Gerar ID de correlação para esta operação
            self._logger.set_correlation_id()
            self._logger.info("Starting user update", user_id=user_id, name=d
to.name, email=dto.email)

            try:
                # Check if user exists
                existing_user = await self._uow.users.find_by_id(user_id)
                if not existing_user:
                    raise UserNotFoundException(str(user_id))

                # Create updated user entity
                updated_user = User.create(name=dto.name, email=dto.email)
                updated_user.id = user_id  # Preserve the ID

                # Create domain service
                domain_service = self._domain_service_factory.create_user_dom
ain_service(
                    self._uow.users
                )
                # Validate update
                await domain_service.validate_user_update(user_id, updated_us
er)

                self._logger.info("User update validation passed", user_id=us
er_id)

                # Save updated user
                saved_user = await self._uow.users.save(updated_user)
                await self._uow.commit()

                self._logger.info("User updated successfully",
                             user_id=saved_user.id,
                             name=saved_user.name.value,
                             email=saved_user.email.value)

                return saved_user

            except (UserValidationException, UserNotFoundException) as e:
                if isinstance(e, UserValidationException):
                    self._logger.error("User update validation failed",
                                 user_id=user_id,
                                 validation_errors=e.validation_errors)
                else:
                    self._logger.error("User not found for update", user_id=u
ser_id)
                raise

            except DomainException as e:
                self._logger.error("Domain error during user update",
                              user_id=user_id,
                              error_code=e.error_code,
                              message=e.message,
                              details=e.details)
                raise

            except Exception as e:
                self._logger.error("Unexpected error during user update",
                              user_id=user_id,
                              error=str(e))
                raise RuntimeError(f"Failed to update user: {str(e)}")


class GetUserUseCase(BaseUseCase):
    async def execute(self, user_id: int) -> User:
        async with self._uow:
            try:
                self._logger.info("Getting user", user_id=user_id)
                user = await self._uow.users.find_by_id(user_id)

                if not user:
                    raise UserNotFoundException(str(user_id))

                self._logger.info("User retrieved successfully", user_id=user
_id)
                return user

            except UserNotFoundException:
                self._logger.error("User not found", user_id=user_id)
                raise

            except Exception as e:
                self._logger.error("Error getting user", user_id=user_id, err
or=str(e))
                raise RuntimeError(f"Failed to get user: {str(e)}")
```

```
class DeleteUserUseCase(BaseUseCase):
    async def execute(self, user_id: int) -> bool:
        async with self._uow:
            try:
                self._logger.info("Starting user deletion", user_id=user_id)

                # Check if user exists
                existing_user = await self._uow.users.find_by_id(user_id)
                if not existing_user:
                    raise UserNotFoundException(str(user_id))

                # Perform deletion
                success = await self._uow.users.delete(user_id)

                if success:
                    await self._uow.commit()
                    self._logger.info("User deleted successfully", user_id=us
er_id)
                else:
                    self._logger.warning("User deletion failed", user_id=user
_id)

                return success

            except UserNotFoundException:
                self._logger.error("User not found for deletion", user_id=use
r_id)
                raise

            except Exception as e:
                self._logger.error("Error deleting user", user_id=user_id, er
ror=str(e))
                raise RuntimeError(f"Failed to delete user: {str(e)}")

# Factory para criar use cases com dependências configuradas
class UseCaseFactory:
    def __init__(self, composition_root):
        self._composition_root = composition_root

    def create_user_use_case(self) -> CreateUserUseCase:
        return self._composition_root.create_user_use_case

    def list_users_use_case(self) -> ListUsersUseCase:
        return self._composition_root.list_users_use_case

    def update_user_use_case(self) -> UpdateUserUseCase:
        return self._composition_root.update_user_use_case

    def get_user_use_case(self) -> GetUserUseCase:
        return self._composition_root.get_user_use_case

    def delete_user_use_case(self) -> DeleteUserUseCase:
        return self._composition_root.delete_user_use_case
```

23. Arquivo: user_commands.py

```
# src/dev_platform/interface/cli/user_commands.py
import asyncio
import click
from typing import List, Optional
from application.user.dtos import UserCreateDTO
from infrastructure.composition_root import CompositionRoot
from infrastructure.database.unit_of_work import SQLUnitOfWork
from infrastructure.logging.structured_logger import StructuredLogger


class UserCommands:
    def __init__(self):
        self._composition_root = CompositionRoot()

    async def create_user_async(self, name: str, email: str) -> str:
        try:
            use_case = self._composition_root.create_user_use_case
            dto = UserCreateDTO(name=name, email=email)
            user = await use_case.execute(dto)
            return f"User created successfully: ID {user.id}, Name: {user.nam
e}, Email: {user.email}"
        except ValueError as e:
            return f"Validation Error: {e}"
        except Exception as e:
            return f"Error creating user: {e}"

    async def list_users_async(self) -> list:
        try:
            use_case = self._composition_root.list_users_use_case
            users = await use_case.execute()
            if not users:
                return ["No users found"]

            result = []
            for user in users:
                # CORRIGIDO: Acessar .value dos value objects
                result.append(f"ID: {user.id}, Name: {user.name.value}, Email
: {user.email.value}")
            return result
        except Exception as e:
            return [f"Error: {e}"]

    # Adicione os métodos para Update, Get, Delete se necessário, seguindo o
padrão
    # Se você tiver implementado os outros comandos (update, get, delete)
    # Lembre-se de chamar .execute() neles também.
    async def update_user_async(self, user_id: int, name: Optional[str] = Non
e, email: Optional[str] = None) -> str:
        try:
            use_case = self._composition_root.update_user_use_case
            # CORRIGIDO: Chamar o método .execute()
            user_dto = use_case.execute(user_id=user_id, name=name, email=ema
il)
            return f"User {user_id} updated successfully: ID {user_dto.id}, N
ame: {user_dto.name}, Email: {user_dto.email}"
        except Exception as e:
            return f"Error updating user: {e}"

    async def get_user_async(self, user_id: int) -> str:
        try:
            use_case = self._composition_root.get_user_use_case
            # CORRIGIDO: Chamar o método .execute()
            user_dto = use_case.execute(user_id=user_id)
            return f"User found: ID {user_dto.id}, Name: {user_dto.name.value
}, Email: {user_dto.email.value}"
        except Exception as e:
            return f"Error getting user: {e}"

    async def delete_user_async(self, user_id: int) -> str:
        try:
            use_case = self._composition_root.delete_user_use_case
            # CORRIGIDO: Chamar o método .execute()
            use_case.execute(user_id=user_id) # Delete pode não retornar um D
TO
            return f"User {user_id} deleted successfully."
        except Exception as e:
            return f"Error deleting user: {e}"

@click.group()
def cli():
    pass

# COMANDOS CLICK - CADA UM AGORA GERENCIA SEU PRÓPRIO asyncio.run() E LIMPEZA
@cli.command()
@click.option('--name', prompt='User name')
@click.option('--email', prompt='User email')
def create_user(name: str, email: str):
    """Create a new user."""
    commands = UserCommands()

    async def _run_create():
        result = await commands.create_user_async(name, email)
        click.echo(result)

    asyncio.run(_run_create())

@cli.command()
def list_users():
    """List all users."""
    commands = UserCommands()

    async def _run_list():
        results = await commands.list_users_async()
        for line in results:
            click.echo(line)

    asyncio.run(_run_list())

@cli.command()
@click.option('--user-id', type=int, prompt='User ID to update')
@click.option('--name', prompt='New user name (leave empty to keep current)',
 default='', show_default=False)
@click.option('--email', prompt='New user email (leave empty to keep current)
', default='', show_default=False)
def update_user(user_id: int, name: str, email: str):
    """Update an existing user."""
    commands = UserCommands()

    async def _run_update():
        result = await commands.update_user_async(user_id, name if name else
None, email if email else None)
        click.echo(result)

    asyncio.run(_run_update())

@cli.command()
@click.option('--user-id', type=int, prompt='User ID to retrieve')
def get_user(user_id: int):
    """Get a user by ID."""
    commands = UserCommands()

    async def _run_get():
        result = await commands.get_user_async(user_id)
        click.echo(result)

    asyncio.run(_run_get())

@cli.command()
@click.option('--user-id', type=int, prompt='User ID to delete')
def delete_user(user_id: int):
    """Delete a user by ID."""
    commands = UserCommands()

    async def _run_delete():
        result = await commands.delete_user_async(user_id)
        click.echo(result)

    asyncio.run(_run_delete())
```

24. Arquivo: value_objects.py

```
# ./src/dev_platform/domain/user/value_objects.py
from dataclasses import dataclass
import re


@dataclass(frozen=True)
class Email:
    value: str

    def __post_init__(self):
        if not self._is_valid():
            raise ValueError("Invalid email format")

    def _is_valid(self) -> bool:
        pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
        return bool(re.match(pattern, self.value))

@dataclass(frozen=True)
class UserName:
    value: str

    def __post_init__(self):
        if not self.value or len(self.value) < 3:
            raise ValueError("Name must be at least 3 characters long")
        if len(self.value) > 100:
            raise ValueError("Name cannot exceed 100 characters")
```