Índice

Compilação de Arquivos - stake_file

Arquivo: composition_root.py

Caminho: composition_root.py
Tamanho: 3693 bytes
Codificação: ISO-8859-1

```
# 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
)


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

    def __init__(self, config: dict = None):
        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:
        return self._domain_service_factory

    # Use Cases - CORRIGIDO: Adicionando domain_service_factory onde
necessÃ¡rio
    def create_user_use_case(self) -> CreateUserUseCase:
        return CreateUserUseCase(
            uow=self.uow,
            logger=self._logger,
            domain_service_factory=self.domain_service_factory  # ADICIONADO
        )

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

    def update_user_use_case(self) -> UpdateUserUseCase:
        return UpdateUserUseCase(
            uow=self.uow,
            logger=self._logger,
            domain_service_factory=self.domain_service_factory  # ADICIONADO
        )


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

    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 = self._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_repository)

    # Utility methods for specific configurations
    def create_enterprise_user_domain_service(self, user_repository) ->
UserDomainService:
        """
        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
        )
```

Arquivo: config.py

Caminho: config.py
Tamanho: 6538 bytes
Codificação: utf-8

```
# 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


class Configuration():
    """Classe avançada para gerenciar configurações da aplicação."""
    _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'):
            self._initialized = False

        if not self._initialized:
            self._environment = os.getenv("ENVIRONMENT", "production") #
Garante 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
ambiente."""
        # Determina o nome do arquivo .env com base no ambiente
        dotenv_path = f".env.{self._environment}"

        # Carrega as variáveis do arquivo .env específico do ambiente
        # override=True garante que variáveis do .env sobrescrevam variáveis
de ambiente existentes
```

```
        load_dotenv(dotenv_path=dotenv_path, override=True)

        # Se for produção e o arquivo .env.production não for usado,
        # ou se variáveis de ambiente já são definidas externamente,
        # load_dotenv() sem path carrega do .env padrão se existir, ou do
ambiente
        # Isso é mais uma garantia, mas o principal é o dotenv_path
        if self._environment == "production" and not
os.path.exists(dotenv_path):
             print(f"Aviso: .env.{self._environment} não encontrado.
Assumindo variáveis de ambiente serão configuradas externamente para
produção.")


    def _load_config_file(self):
        """Carrega e mescla configurações de arquivos JSON específicos do
ambiente."""
        config_file_path = f"config.{self._environment}.json"
        if os.path.exists(config_file_path):
            try:
                with open(config_file_path, 'r') as f:
                    environment_config = json.load(f)
                    self._config.update(environment_config)
            except Exception as e:
                warnings.warn(f"Erro ao carregar o arquivo de configuração
{config_file_path}: {e}")

        # Carrega configs padrão se não houver configs específicas do ambiente
já carregadas

        # ou se você tiver um arquivo de config padrão.
        # Por simplicidade, assumimos que as configurações são principalmente
via .env agora.
        # Se você tiver um config.json padrão, você carregaria ele primeiro e
depois faria o update com o específico.

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

    def get(self, key: str, default: Any = None) -> Any:
        """Obtém um valor de configuração, preferindo variáveis de
ambiente."""
        env_value = os.getenv(key.upper().replace('.', '_')) # Converte key
para formato de var de ambiente (ex: logging.level -> LOGGING_LEVEL)
        if env_value is not None:
            return env_value
        return self._config.get(key, default)

    # Remova os métodos get_database_url, get_sync_database_url e
get_async_database_url
    # e acesse DATABASE_URL diretamente via get() ou os.getenv()
    # Exemplo: CONFIG.get("DATABASE_URL") ou os.getenv("DATABASE_URL")

    # Mantenha _ensure_async_driver se ainda precisar dele para transformar
```

```
URLs
    def _ensure_async_driver(self, url: str) -> str:
        # Sua implementação existente de _ensure_async_driver
        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, garantindo driver assíncrono."""
        url = self.get("DATABASE_URL") # Obtém do .env ou ambiente
        if not url:
            raise ConfigurationException("DATABASE_URL is not configured.")
        return self._ensure_async_driver(url)

    @property
    def sync_database_url(self) -> str:
        """Retorna a URL do banco de dados, sem garantir driver assíncrono."""
        url = self.get("DATABASE_URL") # Obtém do .env ou ambiente
        if not url:
            raise ConfigurationException("DATABASE_URL is not configured.")
        return url

    def get_config(self) -> Dict[str, Any]:
        """Retorna todas as configurações como um dicionário."""
        return self.config

    # def get(self, path: str, default: Any = None) -> Any:
    #     """Obtém um valor de configuração por caminho pontilhado, ex:
'database.url'"""
    #     keys = path.split(".")
    #     value = self.config

    #     for key in keys:
    #         if isinstance(value, dict) and key in value:
    #             value = value[key]

    #         else:
    #             return default

    #     return value

    def is_development(self) -> bool:
        """Verifica se está em ambiente de desenvolvimento."""
        return self.environment == "development"

    def is_production(self) -> bool:
        """Verifica se está em ambiente de produção."""
        return self.environment == "production"

    def is_test(self) -> bool:
        """Verifica se está em ambiente de teste."""
        return self.environment == "test"


# Instância Singleton para uso fácil em outros módulos
```

```
CONFIG = Configuration()
DATABASE_URL = CONFIG.get("DATABASE_URL") # CONFIG.database_url()
```

Arquivo: dtos.py

Caminho: dtos.py
Tamanho: 1387 bytes
Codificação: utf-8

```
#   src/dev_platform/application/user/dtos.py
from pydantic import BaseModel, validator


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

    @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
dependência circular
        return User.create(name=self.name, email=self.email)

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

    @validator('name')
    def validate_name(cls, v):
        if not v or len(v.strip()) < 3:
            raise ValueError('Name must be at least 3 characters')
        return v.strip()


    @validator('email')
    def validate_email(cls, v):
        # Validação básica antes de criar Value Object
        return v.lower().strip()

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

    @validator('name')
    def validate_name(cls, v):
        if not v or len(v.strip()) < 3:
            raise ValueError('Name must be at least 3 characters')
        return v.strip()
    @validator('email')
    def validate_email(cls, v):
        # Validação básica antes de criar Value Object
        return v.lower().strip()
```

Arquivo: entities.py

Caminho: entities.py
Tamanho: 570 bytes
Codificação: ascii

```
#   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)
```

Arquivo: exceptions.py

Caminho: exceptions.py
Tamanho: 9820 bytes
Codificação: utf-8

```
#   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: 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:
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:
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_exception else None
        }


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

    def __init__(self, operation: str, reason: str, original_exception: 
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_exception: 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, 
original_exception: Exception = None):
        self.repository_name = repository_name
        self.operation = operation
        self.reason = reason
        super().__init__(
            message=f"Repository '{repository_name}' {operation} failed: 
{reason}",
            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: 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: 
{corruption_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: 
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
validation_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":
reason}

        )


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

    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
UserValidationException instead."""

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

Arquivo: main.py

Caminho: main.py
Tamanho: 118 bytes
Codificação: ascii

```
#   src/dev_platform/main.py
from interface.cli.user_commands import cli


if __name__ == "__main__":
    cli()
```

Arquivo: models.py

Caminho: models.py
Tamanho: 403 bytes
Codificação: ascii

```
#   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)
```

Arquivo: ports.py

Caminho: ports.py
Tamanho: 1422 bytes
Codificação: ascii

```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
```

Arquivo: repositories.py

Caminho: repositories.py
Tamanho: 11007 bytes
Codificação: ascii

```
# 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,
**context):
        """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.orig).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
```

```python
                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
                )

        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
                )

        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
                )

        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))
            )
            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
                )

        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
                )

        async def count(self) -> int:
            """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
                )


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

        @staticmethod
        def handle_sqlalchemy_error(operation: str, error: SQLAlchemyError,
    **context):
            """Handle SQLAlchemy specific errors."""
            if isinstance(error, IntegrityError):
                if "email" in str(error.orig).lower() and "unique" in
    str(error.orig).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
            )
```

Arquivo: services.py

Caminho: services.py
Tamanho: 14693 bytes
Codificação: utf-8

```
# 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: 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_domains)

    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
domains: {', '.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("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
-'àáâãèéêìíîòóôõùúûçÀÁÂÃÈÉÊÌÍÎÒÓÔÕÙÚÛÇ")  # Carregue de config externa
        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.allowed_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 AM
- 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:
List[ValidationRule] = None):
        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
available"
            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
rules
        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
constraints: {str(e)}"


        if validation_errors:
            raise UserValidationException(validation_errors)

    async def validate_business_domain_rules(self, user: User,
domain_whitelist: 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: 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/database
            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) ->
UserAnalyticsService:
        """Create a UserAnalyticsService."""
        return UserAnalyticsService(user_repository)
```

Arquivo: session.py

Caminho: session.py
Tamanho: 4156 bytes
Codificação: utf-8

```
# 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_sessionmaker
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")
        print(async_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
precisa 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
```

Arquivo: structured_logger.py

Caminho: structured_logger.py
Tamanho: 2574 bytes
Codificação: utf-8

```
# 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 
correlação 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} | 
{extra}",
            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)
```

Arquivo: unit_of_work.py

Caminho: unit_of_work.py
Tamanho: 1785 bytes
Codificação: MacRoman

```
# src/dev_platform/infrastructure/database/unit_of_work.py
from typing import Optional
from sqlalchemy.ext.asyncio import AsyncSession
from application.user.ports import UnitOfWork as AbstractUnitOfWork
from infrastructure.database.session import db_manager
from infrastructure.database.repositories import SQLUserRepository


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

    async def __aenter__(self):
        # Usar o gerenciador de sess√µes
        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_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()
```

Arquivo: use_cases.py

Caminho: use_cases.py
Tamanho: 9997 bytes
Codificação: utf-8

```
# 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_factory: 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_domain_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",
email=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(users))
                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_factory: 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=dto.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_domain_service(
                    self._uow.users

                )

                # Validate update
                await domain_service.validate_user_update(user_id,
updated_user)

                self._logger.info("User update validation passed",
user_id=user_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=user_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,
error=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=user_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=user_id)
                raise

            except Exception as e:
                self._logger.error("Error deleting user", user_id=user_id,
error=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()
```

Arquivo: user_commands.py

Caminho: user_commands.py
Tamanho: 2084 bytes
Codificação: utf-8

```
# src/dev_platform/interface/cli/user_commands.py
import asyncio
import click
from application.user.dtos import UserCreateDTO
from infrastructure.composition_root import CompositionRoot
from infrastructure.config import CONFIG

class UserCommands:
    def __init__(self, config: dict = None):

        # CORRIGIDO: Passar configuração para o CompositionRoot
        self._composition_root = CompositionRoot(config=CONFIG.get_config())

    async def create_user(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: {user.name.value} ({user.email.value})
with ID: {user.id}"  # CORRIGIDO: .value
        except ValueError as e:
            return f"Validation Error: {e}"
        except Exception as e:
            return f"Error: {e}"

    async def list_users(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}"]

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

@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()
    result = asyncio.run(commands.create_user(name, email))
    click.echo(result)

@cli.command()
def list_users():
    """List all users."""
    commands = UserCommands()
    results = asyncio.run(commands.list_users())
    for line in results:
        click.echo(line)
```

Arquivo: value_objects.py

Caminho: value_objects.py
Tamanho: 765 bytes
Codificação: ascii

```
# 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")
```