Dependency Injection and Service Setup

Overview

The application implements a robust dependency injection system that manages component lifecycles, service instantiation, and dependency resolution. This system is crucial for:

1. Maintaining clean architecture boundaries
2. Enabling unit testing through dependency mocking
3. Managing resource lifecycles (especially database connections)
4. Ensuring proper service initialization order
5. Supporting different implementation swapping

Detailed Component Analysis

1. Database Session Management

Location: `shared/infrastructure/database.py`

The database session management system ensures proper connection handling and resource cleanup:

```
def get_db() -> Generator[Session, None, None]:
    """
    Database session factory and lifecycle manager.

    Implementation details:
    1. Creates new session from factory
    2. Yields session for dependency injection
    3. Ensures cleanup in finally block

    Features:
    1. Connection pooling
    2. Session scoping
    3. Automatic cleanup
    4. Error handling

    Usage:
    - All repository implementations
    - Direct database access
    - Transaction management
    """
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# Session factory configuration
SessionLocal = sessionmaker(
    bind=engine,
    autocommit=False,
    autoflush=False,
    expire_on_commit=False
)
```

2. Auth Module Dependencies

Location: `auth/presentation/router.py`

Complex dependency graph for authentication components:

```
def get_auth_use_cases(
    db: Session = Depends(get_db)
) -> AuthUseCases:
    """
    Auth module dependency factory.

    Dependency tree:
    AuthUseCases
    └── AuthService
        └── UserRepository
            └── Database Session

    Responsibilities:
    1. Component instantiation
    2. Dependency wiring
    3. Lifecycle management
    4. Resource cleanup

    Usage contexts:
    1. User registration
    2. Login processing
    3. Token validation
    4. Password management
    """
    user_repository = SQLAlchemyUserRepository(db)
    auth_service = AuthService(user_repository)
    return AuthUseCases(auth_service)

def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: Session = Depends(get_db)
) -> User:
    """
    User authentication dependency.

    Security features:
    1. Token validation
    2. User existence check
    3. Permission verification
    4. Session validation

    Error handling:
    1. Invalid tokens
    2. Expired sessions
    3. Missing users
    4. Permission issues
    """
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception

    user = db.query(UserModel).filter(
        UserModel.username == username
    ).first()
    if user is None:
        raise credentials_exception
    return User.from_orm(user)
```

3. Posts Module Dependencies

Location: `posts/presentation/router.py`

Dependency configuration for post management:

```
def get_post_use_cases(
    db: Session = Depends(get_db)
) -> PostUseCases:
    """
    Posts module dependency factory.

    Component hierarchy:
    PostUseCases
    ├── PostService
    │   └── PostRepository
    │       └── Database Session
    └── VoteService (optional)
        └── VoteRepository
            └── Database Session

    Features:
```

```
    1. Lazy initialization
    2. Resource sharing
    3. Circular dependency handling
    4. Error propagation
    """
    post_repository = SQLAlchemyPostRepository(db)
    post_service = PostService(post_repository)
    return PostUseCases(post_service)
```

4. Votes Module Dependencies

Location: `votes/presentation/router.py`

Complex dependency resolution for voting system:

```
def get_vote_use_cases(
    db: Session = Depends(get_db)
) -> VoteUseCases:
    """
    Votes module dependency factory.

    Dependency graph:
    VoteUseCases
    ├── VoteService
    │   ├── VoteRepository
    │   │   └── Database Session
    │   └── PostService
    │       └── PostRepository
    │           └── Database Session
    └── NotificationService (optional)
        └── MessageQueue

    Integration points:
    1. Post updates
    2. User notifications
    3. Analytics tracking
    4. Cache invalidation
    """
    vote_repository = SQLAlchemyVoteRepository(db)
    post_repository = SQLAlchemyPostRepository(db)
    post_service = PostService(post_repository)
    vote_service = VoteService(vote_repository, post_service)
    return VoteUseCases(vote_service)
```

Advanced Features

1. Scoped Dependencies

```python
def get_request_scoped_repository(
    request_id: str,
    db: Session = Depends(get_db)
) -> Repository:
    """
    Request-scoped dependency management.

    Features:
    1. Per-request instances
    2. Resource isolation
    3. Request tracking
    4. Automatic cleanup

    Use cases:
    1. Request tracing
    2. Transaction isolation
    3. Rate limiting
    4. Audit logging
    """
    return RequestScopedRepository(db, request_id)
```

2. Conditional Dependencies

```python
def get_cache_service(
    config: Settings = Depends(get_settings)
) -> CacheService:
    """
    Environment-aware dependency resolution.

    Selection criteria:
    1. Environment (dev/prod)
    2. Configuration settings
    3. Available resources
    4. Performance requirements

    Implementations:
    1. Redis for production
    2. In-memory for development
    3. Null cache for testing
    """
    if config.environment == "production":
        return RedisCacheService(config.redis_url)
    return InMemoryCacheService()
```

Testing Support

1. Dependency Mocking

```
def get_test_dependencies() -> Dict[str, Any]:
    """
    Test dependency configuration.

    Features:
    1. Mock repositories
    2. In-memory databases
    3. Fake services
    4. Test configurations

    Usage:
    1. Unit tests
    2. Integration tests
    3. Performance tests
    4. Behavior verification
    """
    db = create_test_database()
    return {
        "db": db,
        "user_repository": MockUserRepository(),
        "post_repository": MockPostRepository(),
        "vote_repository": MockVoteRepository()
    }
```

Performance Considerations

1. **Dependency Resolution**:

    - Lazy loading
    - Instance caching
    - Resource pooling
    - Cleanup optimization

2. **Resource Management**:

    - Connection pooling
    - Instance reuse
    - Memory management
    - Resource limits

3. **Caching Strategy**:

    - Dependency results
    - Configuration values
    - Service instances
    - Query results

Best Practices

1. **Dependency Organization**:

   - Clear hierarchy
   - Single responsibility
   - Interface segregation
   - Dependency inversion

2. **Resource Lifecycle**:

   - Proper initialization
   - Cleanup handling
   - Error recovery
   - Resource limits

3. **Testing Support**:

   - Easy mocking
   - Isolated testing
   - Configuration overrides
   - Behavior verification

Benefits of This Approach

1. **Testability**:

   - Easy to mock dependencies
   - Services can be tested in isolation
   - No global state

2. **Flexibility**:

   - Easy to swap implementations
   - Dependencies are explicit
   - Clear service boundaries

3. **Maintainability**:

   - Clear dependency graph
   - Centralized dependency management
   - Easy to modify service creation

4. **Scoped Resources**:

   - Database sessions are properly managed
   - Resources are cleaned up automatically
   - Prevents memory leaks

Example Usage in Routes

```
@router.post("/posts/", response_model=Post)
def create_post(
    post: PostCreate,
    current_user: User = Depends(get_current_user),
    post_use_cases: PostUseCases = Depends(get_post_use_cases)
):
    return post_use_cases.create_post(post, current_user)

@router.post("/vote/", response_model=Vote)
def vote(
    vote: VoteCreate,
    current_user: User = Depends(get_current_user),
    vote_use_cases: VoteUseCases = Depends(get_vote_use_cases)
):
    return vote_use_cases.vote_post(vote, current_user)
```

Testing with Dependencies

~~~
def test_create_post():
    # Create mock dependencies
    mock_post_repository = MockPostRepository()
    mock_post_service = PostService(mock_post_repository)
    post_use_cases = PostUseCases(mock_post_service)

    # Test the use case
    post = PostCreate(title="Test", content="Content")
    user = User(id=1, email="test@test.com", username="test")
    result = post_use_cases.create_post(post, user)

    # Assert results
    assert result.title == "Test"
    assert result.owner_id == 1

# Application Setup and Dependency Injection

## 1. Application Entry Point

The main application entry point (`main.py`) sets up FastAPI and configures all
dependencies:

```python
# main.py
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from .auth.presentation.router import router as auth_router
from .posts.presentation.router import router as posts_router
from .votes.presentation.router import router as votes_router
from .shared.infrastructure.database import engine, Base

# Create database tables
Base.metadata.create_all(bind=engine)

app = FastAPI()

# CORS middleware configuration
~~~

```python
origins = ["*"]  # In production, replace with specific origins

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Include routers
app.include_router(auth_router)
app.include_router(posts_router)
app.include_router(votes_router)
```

2. Shared Infrastructure

2.1 Database Configuration (`shared/infrastructure/database.py`)

```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from sqlalchemy.orm import Session
from typing import Generator
from ..config import settings

SQLALCHEMY_DATABASE_URL = f'postgresql://{settings.database_username}:{settings.database_password}@{settings.database_hostname}:{settings.database_port}/{settings.database_name}'

engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db() -> Generator[Session, None, None]:
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

This setup provides:

1. Database connection configuration
2. Session management
3. Base class for models
4. Database dependency injection

3. Layer Implementation

3.1 Domain Layer

The domain layer contains the core business logic and is framework-independent:

3.1.1 Entities

- Define business objects (`User`, `Post`, `Vote`)
- Contain validation rules
- Use Pydantic for data validation
- Independent of persistence

3.1.2 Repository Interfaces

- Define data access contracts
- Independent of database implementation
- Use domain entities
- Enforce business rules

3.1.3 Domain Services

- Implement business rules
- Independent of use cases
- Operate on domain entities
- Handle core operations

3.2 Application Layer

The application layer orchestrates the domain layer:

3.2.1 Use Cases

- Coordinate domain objects
- Handle application flow
- Implement user stories
- Manage transactions

3.3 Infrastructure Layer

The infrastructure layer implements technical details:

3.3.1 Database Models

- Map domain entities to database
- Handle persistence
- Define relationships
- Manage constraints

3.3.2 Repository Implementations

- Implement repository interfaces
- Handle database operations
- Convert between models and entities
- Manage transactions

3.4 Presentation Layer

The presentation layer handles HTTP concerns:

3.4.1 API Routes

- Define endpoints
- Handle HTTP requests/responses
- Manage authentication
- Inject dependencies

4. Dependency Flow

4.1 Request Flow

1. HTTP request arrives at router
2. FastAPI injects dependencies:
    - Database session
    - Current user (if authenticated)
    - Use cases
3. Use case orchestrates:
    - Domain services
    - Repository operations
4. Repository implements:
    - Database operations
    - Entity conversion

4.2 Module Dependencies

Auth Module

```
Router
└── AuthUseCases
    ├── UserRepository
    │   └── Database Session
    └── AuthService
        └── UserRepository
```

Posts Module

```
Router
└── PostUseCases
    └── PostService
        └── PostRepository
            └── Database Session
```

Votes Module

```
Router
└── VoteUseCases
    └── VoteService
        ├── VoteRepository
        │   └── Database Session
        └── PostService
            └── PostRepository
                └── Database Session
```

5. Dependency Injection Points

5.1 FastAPI Dependencies

1. **Database Session**

```python
def get_db() -> Generator[Session, None, None]:
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

2. **Current User**

```python
def get_current_user(
    token: Annotated[str, Depends(oauth2_scheme)],
    db: Session = Depends(get_db)
) -> User:
    # Verify token and return user
```

3. **Use Cases**

```python
def get_auth_use_cases(db: Session = Depends(get_db)) -> AuthUseCases:
    auth_service = AuthService(user_repository=SQLAlchemyUserRepository(db))
    return AuthUseCases(
        user_repository=SQLAlchemyUserRepository(db, auth_service),
        auth_service=auth_service
    )
```

5.2 Constructor Injection

1. **Repository Dependencies**

```python
class SQLAlchemyUserRepository:
    def __init__(self, db: Session, auth_service: AuthService):
        self.db = db
        self.auth_service = auth_service
```

2. **Service Dependencies**

```python
class AuthService:
    def __init__(self, user_repository: UserRepository):
        self.user_repository = user_repository
```

3. **Use Case Dependencies**

```python
class PostUseCases:
    def __init__(self, post_service: PostService):
        self.post_service = post_service
```

6. Benefits and Best Practices

6.1 Clean Architecture Benefits

1. **Separation of Concerns**
   - Each layer has a single responsibility
   - Changes in one layer don't affect others
   - Easy to test each layer in isolation
2. **Dependency Inversion**
   - Domain layer defines interfaces
   - Implementation details in outer layers
   - Easy to swap implementations
3. **Testability**
   - Can mock repositories and services
   - Business logic is framework-independent

  - Can test use cases without database

6.2 Best Practices

1. **Dependency Management**
    - Use constructor injection
    - Depend on abstractions
    - Keep dependencies minimal
    - Use factory functions
2. **Layer Isolation**
    - No circular dependencies
    - Domain layer is independent
    - Infrastructure depends on interfaces
    - Presentation handles HTTP only
3. **Testing**
    - Mock external dependencies
    - Test business rules in isolation
    - Use in-memory repositories
    - Test use cases independently
4. **Error Handling**
    - Domain exceptions for business rules
    - Application exceptions for use cases
    - Infrastructure exceptions for technical issues
    - Presentation layer maps to HTTP codes
5. **Configuration**
    - Externalize settings
    - Use environment variables
    - Configure at startup
    - Validate configuration

7. Layer Overview

Our application follows clean architecture principles with four distinct layers:

7.1 Domain Layer (`/domain`)

The core business logic layer containing:

- Business entities
- Repository interfaces
- Domain services
- Value objects

- Business rules

7.2 Application Layer (`/application`)

The use case orchestration layer containing:

- Use case implementations
- DTOs
- Input/output ports
- Application services

7.3 Infrastructure Layer (`/infrastructure`)

The technical details layer containing:

- Repository implementations
- Database models
- External service integrations
- Framework-specific code

7.4 Presentation Layer (`/presentation`)

The user interface layer containing:

- API routes
- Request/response models
- Controllers
- View models

8. Detailed Layer Implementation

8.1 Domain Layer Implementation

8.1.1 Entities

Auth Module Entities

```
# auth/domain/entities/user.py
class UserBase(BaseModel):
    email: EmailStr
    username: str

class UserCreate(UserBase):
    password: str

class User(UserBase):
    id: int
    created_at: datetime

    class Config:
        from_attributes = True
```

Posts Module Entities

```python
# posts/domain/entities/post.py
class PostBase(BaseModel):
    title: str
    content: str
    published: bool = True

class PostCreate(PostBase):
    pass

class Post(PostBase):
    id: int
    created_at: datetime
    owner_id: int
    owner_username: str
    votes: int = 0

    class Config:
        from_attributes = True
```

Votes Module Entities

```python
# votes/domain/entities/vote.py
class VoteBase(BaseModel):
    post_id: int
    dir: int  # 1 for upvote, -1 for downvote

class VoteCreate(VoteBase):
    pass

class Vote(VoteBase):
    id: int
    user_id: int
    created_at: datetime

    class Config:
        from_attributes = True
```

8.1.2 Repository Interfaces

Auth Repository Interface

```
# auth/domain/repositories/user_repository.py
class UserRepository(ABC):
    @abstractmethod
    def create(self, user: UserCreate) -> User:
        pass

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

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

    @abstractmethod
    def get_by_username(self, username: str) -> Optional[User]:
        pass
```

Posts Repository Interface

```
# posts/domain/repositories/post_repository.py
class PostRepository(ABC):
    @abstractmethod
    def create(self, post: PostCreate, owner_id: int) -> Post:
        pass

    @abstractmethod
    def get_by_id(self, post_id: int) -> Optional[Post]:
        pass

    @abstractmethod
    def get_all(self, skip: int = 0, limit: int = 10) -> List[Post]:
        pass

    @abstractmethod
    def get_by_owner(self, owner_id: int) -> List[Post]:
        pass

    @abstractmethod
    def update(self, post_id: int, post: PostUpdate, owner_id: int) ->
Optional[Post]:
        pass

    @abstractmethod
    def delete(self, post_id: int, owner_id: int) -> bool:
        pass

    @abstractmethod
    def update_votes(self, post_id: int, vote_count: int) -> Optional[Post]:
        pass
```

Votes Repository Interface

```python
# votes/domain/repositories/vote_repository.py
class VoteRepository(ABC):
    @abstractmethod
    def create(self, vote: VoteCreate, user_id: int) -> Vote:
        pass

    @abstractmethod
    def get_vote(self, post_id: int, user_id: int) -> Optional[Vote]:
        pass

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

    @abstractmethod
    def get_vote_count(self, post_id: int) -> int:
        pass
```

8.1.3 Domain Services

Auth Service

```python
# auth/domain/services/auth_service.py
class AuthService:
    def __init__(self, user_repository: UserRepository):
        self.user_repository = user_repository

    def verify_password(self, plain_password: str, hashed_password: str):
        return pwd_context.verify(plain_password, hashed_password)

    def get_password_hash(self, password: str):
        return pwd_context.hash(password)

    def create_access_token(self, data: dict):
        to_encode = data.copy()
        expire = datetime.utcnow() +
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
        to_encode.update({"exp": expire})
        return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
```

Posts Service

```
# posts/domain/services/post_service.py
class PostService:
    def __init__(self, post_repository: PostRepository):
        self.post_repository = post_repository

    def create_post(self, post: PostCreate, current_user: User) -> Post:
        return self.post_repository.create(post, current_user.id)

    def get_post(self, post_id: int) -> Optional[Post]:
        return self.post_repository.get_by_id(post_id)

    def get_posts(self, skip: int = 0, limit: int = 10) -> List[Post]:
        return self.post_repository.get_all(skip, limit)

    def update_post(self, post_id: int, post: PostUpdate, current_user: User) ->
Optional[Post]:
        return self.post_repository.update(post_id, post, current_user.id)

    def delete_post(self, post_id: int, current_user: User) -> bool:
        return self.post_repository.delete(post_id, current_user.id)
```

Votes Service

```
# votes/domain/services/vote_service.py
class VoteService:
    def __init__(self, vote_repository: VoteRepository, post_service:
PostService):
        self.vote_repository = vote_repository
        self.post_service = post_service

    def vote_post(self, vote: VoteCreate, current_user: User) -> Vote:
        # Check if post exists
        post = self.post_service.get_post(vote.post_id)
        if not post:
            raise HTTPException(status_code=404, detail="Post not found")

        # Get existing vote
        existing_vote = self.vote_repository.get_vote(vote.post_id,
current_user.id)

        if vote.dir == 1:
            if existing_vote:
                raise HTTPException(status_code=409, detail="Already voted")
            return self.vote_repository.create(vote, current_user.id)
        else:
            if not existing_vote:
                raise HTTPException(status_code=404, detail="Vote not found")
            return self.vote_repository.delete(vote.post_id, current_user.id)
```

8.2 Application Layer Implementation

8.2.1 Use Cases

Auth Use Cases

```python
# auth/application/auth_use_cases.py
class AuthUseCases:
    def __init__(self, user_repository: UserRepository, auth_service: AuthService):
        self.user_repository = user_repository
        self.auth_service = auth_service

    def register_user(self, user: UserCreate) -> User:
        if self.user_repository.get_by_email(user.email):
            raise HTTPException(status_code=400, detail="Email already registered")
        return self.user_repository.create(user)

    def authenticate_user(self, username: str, password: str) -> User:
        user = self.user_repository.get_by_username(username)
        if not user or not self.auth_service.verify_password(password, user.password):
            raise HTTPException(status_code=400, detail="Incorrect username or password")
        return user
```

Posts Use Cases

```python
# posts/application/post_use_cases.py
class PostUseCases:
    def __init__(self, post_service: PostService):
        self.post_service = post_service

    def create_post(self, post: PostCreate, current_user: User) -> Post:
        return self.post_service.create_post(post, current_user)

    def get_posts(self, skip: int = 0, limit: int = 10) -> List[Post]:
        return self.post_service.get_posts(skip, limit)

    def update_post(self, post_id: int, post: PostUpdate, current_user: User) -> Post:
        return self.post_service.update_post(post_id, post, current_user)

    def delete_post(self, post_id: int, current_user: User) -> bool:
        return self.post_service.delete_post(post_id, current_user)
```

Votes Use Cases

```
# votes/application/vote_use_cases.py
class VoteUseCases:
    def __init__(self, vote_service: VoteService):
        self.vote_service = vote_service

    def vote_post(self, vote: VoteCreate, current_user: User) -> Vote:
        return self.vote_service.vote_post(vote, current_user)
```

8.3 Infrastructure Layer Implementation

8.3.1 Database Models

Auth Models

```
# auth/infrastructure/models.py
class UserModel(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, nullable=False)
    email = Column(String, nullable=False, unique=True)
    username = Column(String, nullable=False, unique=True)
    password = Column(String, nullable=False)
    created_at = Column(DateTime(timezone=True), nullable=False,
server_default=text('now()'))
```

Posts Models

```
# posts/infrastructure/models.py
class PostModel(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True, nullable=False)
    title = Column(String, nullable=False)
    content = Column(String, nullable=False)
    published = Column(Boolean, server_default='TRUE', nullable=False)
    created_at = Column(DateTime(timezone=True), nullable=False,
server_default=text('now()'))
    owner_id = Column(Integer, ForeignKey("users.id", ondelete="CASCADE"),
nullable=False)
    votes = Column(Integer, server_default='0', nullable=False)
    owner = relationship("UserModel", back_populates="posts")
```

Votes Models

```python
# votes/infrastructure/models.py
class VoteModel(Base):
    __tablename__ = "votes"

    id = Column(Integer, primary_key=True, nullable=False)
    user_id = Column(Integer, ForeignKey("users.id", ondelete="CASCADE"), nullable=False)
    post_id = Column(Integer, ForeignKey("posts.id", ondelete="CASCADE"), nullable=False)
    created_at = Column(DateTime(timezone=True), nullable=False, server_default=text('now()'))
```

8.3.2 Repository Implementations

Auth Repository Implementation

```python
# auth/infrastructure/user_repository_impl.py
class SQLAlchemyUserRepository(UserRepository):
    def __init__(self, db: Session, auth_service: AuthService):
        self.db = db
        self.auth_service = auth_service

    def create(self, user: UserCreate) -> User:
        hashed_password = self.auth_service.get_password_hash(user.password)
        db_user = UserModel(
            email=user.email,
            username=user.username,
            password=hashed_password
        )
        self.db.add(db_user)
        self.db.commit()
        self.db.refresh(db_user)
        return User.from_orm(db_user)
```

Posts Repository Implementation

```python
# posts/infrastructure/post_repository_impl.py
class SQLAlchemyPostRepository(PostRepository):
    def __init__(self, db: Session):
        self.db = db

    def create(self, post: PostCreate, owner_id: int) -> Post:
        db_post = PostModel(
            **post.dict(),
            owner_id=owner_id
        )
        self.db.add(db_post)
        self.db.commit()
        self.db.refresh(db_post)
        return Post.from_orm(db_post)

    def get_all(self, skip: int = 0, limit: int = 10) -> List[Post]:
        posts = self.db.query(PostModel)\
            .order_by(desc(PostModel.created_at))\
            .offset(skip)\
            .limit(limit)\
            .all()
        return [Post.from_orm(post) for post in posts]
```

Votes Repository Implementation

```
# votes/infrastructure/vote_repository_impl.py
class SQLAlchemyVoteRepository(VoteRepository):
    def __init__(self, db: Session):
        self.db = db

    def create(self, vote: VoteCreate, user_id: int) -> Vote:
        db_vote = VoteModel(
            post_id=vote.post_id,
            user_id=user_id
        )
        self.db.add(db_vote)
        self.db.commit()
        self.db.refresh(db_vote)
        return Vote.from_orm(db_vote)

    def get_vote_count(self, post_id: int) -> int:
        return self.db.query(VoteModel)\
            .filter(VoteModel.post_id == post_id)\
            .count()
```

8.4 Presentation Layer Implementation

8.4.1 API Routes

Auth Routes

```
# auth/presentation/router.py
@router.post("/register", response_model=User)
def register(user: UserCreate, db: Session = Depends(get_db)):
    auth_service = AuthService(user_repository=SQLAlchemyUserRepository(db))
    use_cases = AuthUseCases(
        user_repository=SQLAlchemyUserRepository(db, auth_service),
        auth_service=auth_service
    )
    return use_cases.register_user(user)
```

Posts Routes

```python
# posts/presentation/router.py
@router.post("/", response_model=Post)
def create_post(
    post: PostCreate,
    current_user: User = Depends(get_current_user),
    post_use_cases: PostUseCases = Depends(get_post_use_cases)
):
    return post_use_cases.create_post(post, current_user)
```

Votes Routes

```python
# votes/presentation/router.py
@router.post("/", response_model=Vote)
def vote(
    vote: VoteCreate,
    current_user: User = Depends(get_current_user),
    vote_use_cases: VoteUseCases = Depends(get_vote_use_cases)
):
    return vote_use_cases.vote_post(vote, current_user)
```

9. Dependency Flow

9.1 Authentication Flow

```
Router (Presentation)
└── AuthUseCases (Application)
    ├── UserRepository (Domain Interface)
    │   └── Database Session
    └── AuthService (Domain)
        └── UserRepository (Domain Interface)
```

9.2 Posts Flow

```
Router (Presentation)
└── PostUseCases (Application)
    └── PostService (Domain)
        └── PostRepository (Domain Interface)
            └── Database Session
```

9.3 Votes Flow

```
Router (Presentation)
└── VoteUseCases (Application)
    └── VoteService (Domain)
        ├── VoteRepository (Domain Interface)
        │   └── Database Session
        └── PostService (Domain)
            └── PostRepository (Domain Interface)
                └── Database Session
```

9.4 Dependencies Injected at Each Layer

1. **Presentation Layer** receives:
    - Database session (from FastAPI's dependency injection)
    - Current user (from auth middleware)
    - Use cases (constructed in route handlers)
2. **Application Layer** (Use Cases) receives:
    - Domain services (injected via constructor)
    - Repository interfaces (through domain services)
3. **Domain Layer** receives:
    - Repository interfaces (injected into services)
    - No external dependencies
4. **Infrastructure Layer** receives:
    - Database session (injected via constructor)
    - Domain services (when needed, like auth_service in user repository)

10. Benefits of This Implementation

1. **Separation of Concerns**
    - Each layer has a single responsibility
    - Changes in one layer don't affect others
    - Easy to test each layer in isolation
2. **Dependency Inversion**
    - Domain layer defines interfaces
    - Implementation details are in outer layers
    - Easy to swap implementations
3. **Testability**

- Can mock repositories and services
- Business logic is independent of frameworks
- Can test use cases without database

4. **Maintainability**

   - Clear boundaries between layers
   - Easy to understand dependencies
   - Consistent dependency flow

5. **Flexibility**

   - Can change database without affecting business logic
   - Can add new features by extending interfaces
   - Can modify presentation layer without touching domain logic