# Clean Architecture Overview

This document provides a comprehensive overview of the clean architecture implementation in our project. The project follows Robert C. Martin's Clean Architecture principles to create a maintainable, testable, and scalable codebase.

## Project Structure

The project is organized into three main modules:

1. `auth` - Handles user authentication and authorization
2. `posts` - Manages blog post creation and management
3. `votes` - Handles post voting system

Each module follows the clean architecture layers:

```
module/
├── application/        # Use cases layer
│   └── *_use_cases.py
├── domain/             # Business rules and entities
│   ├── entities/       # Business objects
│   ├── repositories/   # Abstract repository interfaces
│   └── services/       # Domain services
├── infrastructure/     # External implementations
│   ├── models.py       # Database models
│   └── *_impl.py      # Repository implementations
└── presentation/       # Controllers/Routes
    └── router.py       # FastAPI routes
```

## Clean Architecture Principles

1. **Independence of Frameworks**: The core business logic (domain layer) is independent of frameworks. For example, our domain entities are pure Python classes, not tied to SQLAlchemy or any other framework.

2. **Dependency Rule**: Dependencies only point inwards. Outer layers can depend on inner layers, but not vice versa:
   - Presentation layer → Application layer → Domain layer
   - Infrastructure layer → Domain layer (implements interfaces)

3. **Abstraction at Boundaries**: We use interfaces (abstract classes) at boundaries between layers. For example, `PostRepository` is an abstract interface implemented by `SQLAlchemyPostRepository`.

## Key Components

### Domain Layer

- Contains business rules and entities
- No dependencies on external frameworks
- Defines repository interfaces
- Contains domain services for business logic

### Application Layer

- Contains use cases that orchestrate the flow of data
- Depends on domain layer
- Independent of external concerns (databases, UI, etc.)

### Infrastructure Layer

- Implements repository interfaces
- Contains database models and ORM configurations
- Handles external concerns (database, external services)

### Presentation Layer

- Contains FastAPI routes and controllers
- Handles HTTP requests and responses
- Uses dependency injection for use cases

## Benefits of This Architecture

1. **Testability**: Business logic can be tested without external dependencies
2. **Maintainability**: Clear separation of concerns makes code easier to maintain
3. **Flexibility**: Easy to swap implementations (e.g., change database)
4. **Independence**: Business logic is framework-agnostic

# 1. Introduction

Our application follows Clean Architecture principles to create a maintainable, testable, and scalable codebase. This document provides an overview of how Clean Architecture is implemented across all modules.

# 2. Architecture Layers

## 2.1 Domain Layer

The innermost layer containing business logic and rules.

#### Key Components

- **Entities**: Core business objects

```
class UserBase(BaseModel):
    email: EmailStr
    username: str
```

- **Repository Interfaces**: Data access contracts

```
class UserRepository(ABC):
    @abstractmethod
    def create(self, user: UserCreate) -> User:
        pass
```

- **Domain Services**: Core business logic

```
class AuthService:
    def verify_password(self, plain_password: str, hashed_password: str) ->
bool:
        return self.pwd_context.verify(plain_password, hashed_password)
```

#### Characteristics

- No dependencies on outer layers
- Pure business logic
- Framework-independent
- Highly testable

### 2.2 Application Layer

Orchestrates the flow of data and coordinates domain objects.

#### Key Components

- **Use Cases**: Application-specific business rules

```
class AuthUseCases:
    def register_user(self, user: UserCreate) -> User:
        if self.user_repository.get_by_email(user.email):
            raise UserExistsError()
        return self.user_repository.create(user)
```

- **DTOs**: Data transfer objects

```
class PostResponse(BaseModel):
    id: int
    title: str
    content: str
    votes: int
```

**Characteristics**

- Depends only on domain layer
- Handles use case orchestration
- Maps exceptions to application errors
- Manages transactions

### 2.3 Infrastructure Layer

Implements technical details and external integrations.

**Key Components**

- **Repository Implementations**: Database access

```
class SQLAlchemyUserRepository(UserRepository):
    def create(self, user: UserCreate) -> User:
        db_user = UserModel(**user.dict())
        self.db.add(db_user)
        self.db.commit()
        return User.from_orm(db_user)
```

- **Database Models**: ORM models

```
class UserModel(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    email = Column(String, unique=True)
    username = Column(String, unique=True)
```

**Characteristics**

- Implements interfaces from domain layer
- Handles database operations
- Manages external services
- Handles technical details

### 2.4 Presentation Layer

Handles HTTP concerns and user interface.

**Key Components**

- **API Routes**: HTTP endpoints

```
@router.post("/register")
def register(user: UserCreate, db: Session = Depends(get_db)):
    use_cases = AuthUseCases(user_repository, auth_service)
    return use_cases.register_user(user)
```

- **Request/Response Models**: API schemas

```
class LoginRequest(BaseModel):
    username: str
    password: str
```

**Characteristics**

- Handles HTTP concerns
- Manages authentication
- Formats responses
- Validates requests

# 3. Dependency Flow

## 3.1 Core Principles

1. Dependencies point inward
2. Inner layers know nothing about outer layers
3. Interfaces defined in domain layer
4. Implementations in infrastructure layer

## 3.2 Example Flow

```
Router (Presentation)
└── AuthUseCases (Application)
    ├── UserRepository (Domain Interface)
    │   └── SQLAlchemyUserRepository (Infrastructure)
    └── AuthService (Domain)
```

# 4. Module Organization

## 4.1 Directory Structure

```
src/
├── auth/
│   ├── domain/
│   │   ├── entities/
│   │   ├── repositories/
```

```
│   │   └── services/
│   ├── application/
│   ├── infrastructure/
│   └── presentation/
├── posts/
│   ├── domain/
│   ├── application/
│   ├── infrastructure/
│   └── presentation/
└── votes/
    ├── domain/
    ├── application/
    ├── infrastructure/
    └── presentation/
```

### 4.2 Module Dependencies

```
Auth Module ◄─── Posts Module ◄─── Votes Module
     │                │                │
     └────────────────┴────────────────┘
              Shared Infrastructure
```

## 5. Implementation Benefits

### 5.1 Maintainability

- Clear separation of concerns
- Independent of frameworks
- Easy to modify implementations
- Consistent structure

### 5.2 Testability

- Domain logic easily tested
- Mocking at interface boundaries
- Independent testing of layers
- Integration test support

### 5.3 Flexibility

- Framework independence
- Database independence
- Easy to add features
- Simple to modify

## 6. Best Practices

### 6.1 Code Organization

- Keep domain logic pure
- Use dependency injection
- Follow interface segregation
- Maintain layer isolation

### 6.2 Testing Strategy

```
# Domain Layer Test
def test_password_verification():
    auth_service = AuthService(mock_repository)
    assert auth_service.verify_password("test123", hashed_password)

# Application Layer Test
def test_user_registration():
    use_cases = AuthUseCases(mock_repository, mock_service)
    result = use_cases.register_user(test_user)
    assert result.email == test_user.email

# Integration Test
def test_login_flow():
    response = client.post("/auth/login", json=credentials)
    assert response.status_code == 200
    assert "access_token" in response.json()
```

### 6.3 Error Handling

```
# Domain Error
class DomainError(Exception):
    """Base class for domain errors"""
    pass

# Application Error
class ApplicationError(Exception):
    """Base class for application errors"""
    pass

# Infrastructure Error
class InfrastructureError(Exception):
    """Base class for infrastructure errors"""
    pass
```

### 6.4 Dependency Injection

```
# Service injection
def get_auth_service(db: Session = Depends(get_db)) -> AuthService:
    return AuthService(SQLAlchemyUserRepository(db))
```

```
# Use case injection
def get_auth_use_cases(
    auth_service: AuthService = Depends(get_auth_service)
) -> AuthUseCases:
    return AuthUseCases(auth_service)
```

# 7. Cross-Cutting Concerns

## 7.1 Logging

```
# Domain logging
logger.info("User %s authenticated successfully", user.id)

# Infrastructure logging
logger.error("Database connection failed: %s", str(e))
```

## 7.2 Caching

```
# Application layer caching
@cache(ttl=300)
def get_user_profile(user_id: int) -> UserProfile:
    return self.user_repository.get_profile(user_id)
```

## 7.3 Security

```
# Authentication middleware
async def get_current_user(
    token: str = Depends(oauth2_scheme),
    auth_service: AuthService = Depends(get_auth_service)
) -> User:
    return auth_service.verify_token(token)
```

# 8. Performance Considerations

## 8.1 Database Optimization

- Use appropriate indexes
- Implement connection pooling
- Optimize queries
- Handle N+1 problems

## 8.2 Caching Strategy

- Cache frequently accessed data

- Use Redis for distributed caching
- Implement cache invalidation
- Monitor cache hit rates

### 8.3 API Performance

- Implement pagination
- Use appropriate serialization
- Handle concurrent requests
- Monitor response times

## 9. Monitoring and Maintenance

### 9.1 Health Checks

```
@router.get("/health")
def health_check():
    return {
        "status": "healthy",
        "database": check_database(),
        "cache": check_cache()
    }
```

### 9.2 Metrics

- Request latency
- Error rates
- Database performance
- Cache hit rates

### 9.3 Logging Strategy

- Use structured logging
- Implement log levels
- Track request context
- Monitor error patterns