Posts Module Documentation

Overview

The posts module is the core content management component of the application. It provides a complete system for:

1. Creating and managing blog posts
2. Implementing ownership-based access control
3. Managing post visibility and publication status
4. Integrating with the voting system
5. Handling post metadata and relationships

Detailed Component Analysis

1. Post Repository Interface

Location: `posts/domain/repositories/post_repository.py`

The `PostRepository` defines the contract for post data operations, following the Repository pattern for data access abstraction.

```
class PostRepository(ABC):
    @abstractmethod
    def create(self, post: PostCreate, owner_id: int) -> Post:
        """
        Creates a new post with ownership.

        Implementation requirements:
        1. Validate post data
        2. Set creation timestamp
        3. Associate with owner
        4. Initialize vote count

        Args:
            post (PostCreate): Post data (title, content, published status)
            owner_id (int): ID of the user creating the post

        Returns:
            Post: Created post with generated ID

        Used by:
            - PostService for post creation
            - Direct usage in creation endpoint
        """
        pass

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

```
        """
        Retrieves a specific post by ID.

        Implementation requirements:
        1. Handle non-existent posts
        2. Load related data (owner, vote count)
        3. Apply visibility rules

        Used by:
            - PostService for post retrieval
            - VoteService for vote validation
            - Update/delete operations
        """
        pass

    @abstractmethod
    def get_all(self, skip: int = 0, limit: int = 10) -> List[Post]:
        """
        Retrieves paginated list of posts.

        Features:
        1. Pagination support
        2. Sorting (newest first)
        3. Published posts only
        4. Includes vote counts

        Performance considerations:
        1. Efficient pagination query
        2. Optimized joins
        3. Caching support
        """
        pass

    @abstractmethod
    def update(self, post_id: int, post: PostUpdate, owner_id: int) ->
Optional[Post]:
        """
        Updates existing post with owner verification.

        Security checks:
        1. Ownership validation
        2. Content validation
        3. Concurrency handling

        Business rules:
        1. Only owner can update
        2. Maintain update history
        3. Validate content changes
        """
        pass
```

2. Post Model Implementation

Location: `posts/infrastructure/models.py`

The SQLAlchemy model defines the database structure and relationships:

```python
class PostModel(Base):
    """
    Database representation of a post.

    Table structure:
    - id: Primary key
    - title: Post title (indexed for search)
    - content: Post content (text type for large content)
    - published: Boolean flag for post status
    - created_at: Timestamp with timezone
    - owner_id: Foreign key to users table
    - votes: Counter cache for vote count

    Indexes:
    1. Primary key (id)
    2. Foreign key (owner_id)
    3. Created_at (for sorting)
    4. Title (for search)

    Relationships:
    1. owner: Many-to-One with User
    2. votes: One-to-Many with Votes
    """
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True, nullable=False)
    title = Column(String, nullable=False, index=True)
    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
    )

    # Relationships
    owner = relationship(
        "UserModel",
        back_populates="posts",
```

```
        lazy="joined"
    )
```

3. Post Service Implementation

Location: `posts/domain/services/post_service.py`

The service layer contains business logic for post operations:

```
class PostService:
    """
    Core business logic for post management.

    Responsibilities:
    1. Post lifecycle management
    2. Access control enforcement
    3. Business rule validation
    4. Integration with other services
    """

    def __init__(self, post_repository: PostRepository):
        self.post_repository = post_repository

    def create_post(self, post: PostCreate, owner_id: int) -> Post:
        """
        Creates new post with validation.

        Business rules:
        1. Title length validation
        2. Content format checking
        3. Spam detection
        4. Rate limiting

        Integration points:
        1. User service (owner validation)
        2. Notification service
        3. Search indexing
        """
        # Content validation
        if len(post.title) < 3:
            raise ValueError("Title too short")
        if len(post.content) < 10:
            raise ValueError("Content too short")

        return self.post_repository.create(post, owner_id)

    def update_post(self, post_id: int, post: PostUpdate, owner_id: int) ->
Optional[Post]:
        """
        Updates post with ownership verification.
```

```
        Security checks:
        1. Owner verification
        2. Content validation
        3. Rate limiting

        Update tracking:
        1. Modification timestamp
        2. Edit history
        3. Version control
        """
        existing = self.post_repository.get_by_id(post_id)
        if not existing:
            raise HTTPException(
                status_code=404,
                detail="Post not found"
            )

        if existing.owner_id != owner_id:
            raise HTTPException(
                status_code=403,
                detail="Not authorized to update this post"
            )

        return self.post_repository.update(post_id, post, owner_id)
```

4. Post Use Cases

Location: `posts/application/post_use_cases.py`

Orchestrates the application flow for post operations:

```
class PostUseCases:
    """
    Application use cases for post management.

    Responsibilities:
    1. Request handling
    2. Response formatting
    3. Error handling
    4. Transaction management
    """

    def __init__(self, post_service: PostService):
        self.post_service = post_service

    async def create_post(self, post: PostCreate, current_user: User) -> Post:
        """
        Complete post creation flow.

        Steps:
        1. Validate request data
```

```
        2. Check user permissions
        3. Create post
        4. Handle notifications
        5. Update search index

        Error handling:
        - Invalid data format
        - Permission denied
        - Rate limiting
        - Database errors
        """
        try:
            return self.post_service.create_post(post, current_user.id)
        except ValueError as e:
            raise HTTPException(status_code=400, detail=str(e))
```

5. Post Router Implementation

Location: `posts/presentation/router.py`

FastAPI routes for post operations:

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

    Security:
    1. Authentication required
    2. Rate limiting
    3. Content validation

    Response handling:
    1. 201 Created
    2. 400 Bad Request
    3. 401 Unauthorized
    4. 429 Too Many Requests
    """
    return await post_use_cases.create_post(post, current_user)
```

Advanced Features

1. Post Visibility System

```
def get_visible_posts(self, user_id: Optional[int] = None) -> List[Post]:
    """
    Smart post visibility system.

    Rules:
    1. Published posts visible to all
    2. Draft posts visible to owner
    3. Archived posts hidden
    4. Scheduled posts time-based
    """
    query = self.db.query(PostModel)
    if not user_id:
        query = query.filter(PostModel.published == True)
    else:
        query = query.filter(
            or_(
                PostModel.published == True,
                and_(
                    PostModel.owner_id == user_id,
                    PostModel.published == False
                )
            )
        )
    return query.all()
```

2. Search and Filtering

```
def search_posts(
    self,
    query: str,
    filters: PostFilters,
    pagination: PaginationParams
) -> PostSearchResult:
    """
    Advanced search implementation.

    Features:
    1. Full-text search
    2. Multiple filters
    3. Smart sorting
    4. Result highlighting

    Performance:
    1. Search index
    2. Query optimization
    3. Result caching
    """
    base_query = self.db.query(PostModel)

    # Apply full-text search
```

```
    if query:
        base_query = base_query.filter(
            or_(
                PostModel.title.ilike(f"%{query}%"),
                PostModel.content.ilike(f"%{query}%")
            )
        )

    # Apply filters
    if filters.author_id:
        base_query = base_query.filter(PostModel.owner_id == filters.author_id)

    # Apply sorting
    base_query = base_query.order_by(PostModel.created_at.desc())

    return paginate(base_query, pagination)
```

Integration Points

1. Vote System Integration

- Vote count tracking
- Vote-based sorting
- Popular posts detection

2. User System Integration

- Author information
- Permission checking
- User activity tracking

3. Notification System

- Post creation notifications
- Comment notifications
- Mention notifications

Performance Considerations

1. **Query Optimization**:
    - Efficient joins
    - Index usage
    - Query caching
2. **Content Storage**:
    - Content compression
    - Binary data handling
    - CDN integration

3. **Caching Strategy**:
    - Post content caching
    - Count caching
    - Query result caching

Layer Implementation

1. Domain Layer (`posts/domain/`)

1.1 Entities (`posts/domain/entities/`)

```python
# posts/domain/entities/post.py
from pydantic import BaseModel
from datetime import datetime
from typing import Optional

class PostBase(BaseModel):
    """Base post attributes shared by all post-related schemas"""
    title: str
    content: str
    published: bool = True

class PostCreate(PostBase):
    """Schema for post creation"""
    pass

class Post(PostBase):
    """Schema for post responses"""
    id: int
    created_at: datetime
    owner_id: int
    owner: dict  # User information
    votes: Optional[int] = 0

    class Config:
        from_attributes = True
```

1.2 Repository Interface (`posts/domain/repositories/`)

```python
# posts/domain/repositories/post_repository.py
from abc import ABC, abstractmethod
from typing import List, Optional
from ..entities.post import Post, PostCreate

class PostRepository(ABC):
    """Abstract interface for post data access"""

    @abstractmethod
```

```
    def create(self, post: PostCreate, user_id: int) -> Post:
        """Create a new post"""
        pass

    @abstractmethod
    def get_by_id(self, post_id: int) -> Optional[Post]:
        """Get post by ID"""
        pass

    @abstractmethod
    def get_all(self, limit: int, skip: int) -> List[Post]:
        """Get all posts with pagination"""
        pass

    @abstractmethod
    def update(self, post_id: int, post: PostCreate) -> Post:
        """Update existing post"""
        pass

    @abstractmethod
    def delete(self, post_id: int) -> None:
        """Delete post by ID"""
        pass

    @abstractmethod
    def get_by_user(self, user_id: int) -> List[Post]:
        """Get all posts by user"""
        pass
```

1.3 Domain Services (`posts/domain/services/`)

```
# posts/domain/services/post_service.py
from typing import List, Optional
from ..repositories.post_repository import PostRepository
from ..entities.post import Post, PostCreate
from ...domain.exceptions import PostNotFoundError, UnauthorizedError

class PostService:
    """Core post business logic"""

    def __init__(self, post_repository: PostRepository):
        self.post_repository = post_repository

    def create_post(self, post: PostCreate, user_id: int) -> Post:
        """Create new post with ownership"""
        return self.post_repository.create(post, user_id)

    def get_post(self, post_id: int) -> Post:
        """Get single post by ID"""
        post = self.post_repository.get_by_id(post_id)
        if not post:
```

```
            raise PostNotFoundError(f"Post {post_id} not found")
        return post

    def get_posts(self, limit: int = 10, skip: int = 0) -> List[Post]:
        """Get all posts with pagination"""
        return self.post_repository.get_all(limit, skip)

    def update_post(self, post_id: int, post: PostCreate, user_id: int) -> Post:
        """Update post with ownership check"""
        existing_post = self.get_post(post_id)
        if existing_post.owner_id != user_id:
            raise UnauthorizedError("Not authorized to update this post")
        return self.post_repository.update(post_id, post)

    def delete_post(self, post_id: int, user_id: int) -> None:
        """Delete post with ownership check"""
        existing_post = self.get_post(post_id)
        if existing_post.owner_id != user_id:
            raise UnauthorizedError("Not authorized to delete this post")
        self.post_repository.delete(post_id)
```

2. Application Layer (posts/application/)

2.1 Use Cases (posts/application/post_use_cases.py)

```
# posts/application/post_use_cases.py
from typing import List
from fastapi import HTTPException, status
from ..domain.entities.post import Post, PostCreate
from ..domain.services.post_service import PostService
from ..domain.exceptions import PostNotFoundError, UnauthorizedError

class PostUseCases:
    """Application use cases for posts"""

    def __init__(self, post_service: PostService):
        self.post_service = post_service

    def create_post(self, post: PostCreate, user_id: int) -> Post:
        """Create new post use case"""
        try:
            return self.post_service.create_post(post, user_id)
        except Exception as e:
            raise HTTPException(
                status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
                detail=str(e)
            )

    def get_posts(self, limit: int = 10, skip: int = 0) -> List[Post]:
        """Get all posts use case"""
        return self.post_service.get_posts(limit, skip)
```

```
    def get_post(self, post_id: int) -> Post:
        """Get single post use case"""
        try:
            return self.post_service.get_post(post_id)
        except PostNotFoundError as e:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail=str(e)
            )

    def update_post(self, post_id: int, post: PostCreate, user_id: int) -> Post:
        """Update post use case"""
        try:
            return self.post_service.update_post(post_id, post, user_id)
        except PostNotFoundError as e:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail=str(e)
            )
        except UnauthorizedError as e:
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail=str(e)
            )

    def delete_post(self, post_id: int, user_id: int) -> None:
        """Delete post use case"""
        try:
            self.post_service.delete_post(post_id, user_id)
        except PostNotFoundError as e:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail=str(e)
            )
        except UnauthorizedError as e:
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail=str(e)
            )
```

3. Infrastructure Layer (`posts/infrastructure/`)

3.1 Database Models (`posts/infrastructure/models.py`)

```
# posts/infrastructure/models.py
from sqlalchemy import Column, Integer, String, Boolean, ForeignKey, DateTime
from sqlalchemy.sql.expression import text
from sqlalchemy.orm import relationship
from ...shared.infrastructure.database import Base
```

```
class PostModel(Base):
    """SQLAlchemy model for posts table"""

    __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
    )

    # Relationships
    owner = relationship("UserModel")
    votes = relationship("VoteModel", back_populates="post")
```

3.2 Repository Implementation (`posts/infrastructure/post_repository_impl.py`)

```
# posts/infrastructure/post_repository_impl.py
from sqlalchemy.orm import Session
from sqlalchemy import func
from typing import List, Optional
from ..domain.repositories.post_repository import PostRepository
from ..domain.entities.post import Post, PostCreate
from .models import PostModel, VoteModel

class SQLAlchemyPostRepository(PostRepository):
    """SQLAlchemy implementation of PostRepository"""

    def __init__(self, db: Session):
        self.db = db

    def create(self, post: PostCreate, user_id: int) -> Post:
        """Create new post in database"""
        db_post = PostModel(
            **post.dict(),
            owner_id=user_id
        )
        self.db.add(db_post)
        self.db.commit()
        self.db.refresh(db_post)

        return Post.from_orm(db_post)
```

```
    def get_by_id(self, post_id: int) -> Optional[Post]:
        """Get post by ID with vote count"""
        post = self.db.query(PostModel)\
            .filter(PostModel.id == post_id)\
            .first()

        if post:
            votes = self.db.query(func.count(VoteModel.post_id))\
                .filter(VoteModel.post_id == post_id)\
                .scalar()
            setattr(post, 'votes', votes)

        return Post.from_orm(post) if post else None

    def get_all(self, limit: int, skip: int) -> List[Post]:
        """Get all posts with vote counts"""
        posts = self.db.query(PostModel)\
            .offset(skip)\
            .limit(limit)\
            .all()

        for post in posts:
            votes = self.db.query(func.count(VoteModel.post_id))\
                .filter(VoteModel.post_id == post.id)\
                .scalar()
            setattr(post, 'votes', votes)

        return [Post.from_orm(p) for p in posts]

    def update(self, post_id: int, post: PostCreate) -> Post:
        """Update existing post"""
        db_post = self.db.query(PostModel)\
            .filter(PostModel.id == post_id)

        db_post.update(post.dict())
        self.db.commit()

        return Post.from_orm(db_post.first())

    def delete(self, post_id: int) -> None:
        """Delete post by ID"""
        self.db.query(PostModel)\
            .filter(PostModel.id == post_id)\
            .delete()
        self.db.commit()

    def get_by_user(self, user_id: int) -> List[Post]:
        """Get all posts by user with vote counts"""
        posts = self.db.query(PostModel)\
            .filter(PostModel.owner_id == user_id)\
            .all()

        for post in posts:
```

```
            votes = self.db.query(func.count(VoteModel.post_id))\
                .filter(VoteModel.post_id == post.id)\
                .scalar()
            setattr(post, 'votes', votes)

        return [Post.from_orm(p) for p in posts]
```

4. Presentation Layer (posts/presentation/)

4.1 API Routes (posts/presentation/router.py)

```python
# posts/presentation/router.py
from fastapi import APIRouter, Depends, status
from sqlalchemy.orm import Session
from typing import List, Annotated
from ...shared.infrastructure.database import get_db
from ...auth.domain.entities.user import User
from ...auth.presentation.oauth2 import get_current_user
from ..domain.entities.post import Post, PostCreate
from ..application.post_use_cases import PostUseCases
from ..domain.services.post_service import PostService
from ..infrastructure.post_repository_impl import SQLAlchemyPostRepository

router = APIRouter(
    prefix="/posts",
    tags=["Posts"]
)

@router.post("/", status_code=status.HTTP_201_CREATED, response_model=Post)
def create_post(
    post: PostCreate,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user)
):
    """Create new post endpoint"""
    post_service = PostService(
        post_repository=SQLAlchemyPostRepository(db)
    )
    use_cases = PostUseCases(post_service)
    return use_cases.create_post(post, current_user.id)

@router.get("/", response_model=List[Post])
def get_posts(
    limit: int = 10,
    skip: int = 0,
    db: Session = Depends(get_db)
):
    """Get all posts endpoint"""
    post_service = PostService(
        post_repository=SQLAlchemyPostRepository(db)
    )
```

```
    use_cases = PostUseCases(post_service)
    return use_cases.get_posts(limit, skip)

@router.get("/{id}", response_model=Post)
def get_post(
    id: int,
    db: Session = Depends(get_db)
):
    """Get single post endpoint"""
    post_service = PostService(
        post_repository=SQLAlchemyPostRepository(db)
    )
    use_cases = PostUseCases(post_service)
    return use_cases.get_post(id)

@router.put("/{id}", response_model=Post)
def update_post(
    id: int,
    post: PostCreate,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user)
):
    """Update post endpoint"""
    post_service = PostService(
        post_repository=SQLAlchemyPostRepository(db)
    )
    use_cases = PostUseCases(post_service)
    return use_cases.update_post(id, post, current_user.id)

@router.delete("/{id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_post(
    id: int,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user)
):
    """Delete post endpoint"""
    post_service = PostService(
        post_repository=SQLAlchemyPostRepository(db)
    )
    use_cases = PostUseCases(post_service)
    use_cases.delete_post(id, current_user.id)
```

Post Operations Flow

1. Create Post Flow

1. Client sends POST to `/posts/` with post data
2. Auth middleware validates token
3. Router validates request using `PostCreate` schema
4. Use case creates post with user ownership
5. Repository saves to database

6. Router returns created post

2. Get Posts Flow

1. Client sends GET to `/posts/`
2. Router handles pagination params
3. Use case retrieves posts
4. Repository loads posts with vote counts
5. Router returns post list

3. Update Post Flow

1. Client sends PUT to `/posts/{id}`
2. Auth middleware validates token
3. Use case verifies ownership
4. Repository updates post
5. Router returns updated post

4. Delete Post Flow

1. Client sends DELETE to `/posts/{id}`
2. Auth middleware validates token
3. Use case verifies ownership
4. Repository deletes post
5. Router returns success

Integration Points

1. Auth Module Integration

- User ownership of posts
- Authentication middleware
- User information in responses

2. Votes Module Integration

- Vote counts in post responses
- Vote relationships
- Cascading deletes

Testing

1. Unit Tests

```
def test_create_post():
    post_service = PostService(mock_repository)
    post = PostCreate(
        title="Test Post",
        content="Test Content"
```

```
    )
    result = post_service.create_post(post, user_id=1)
    assert result.title == post.title
    assert result.owner_id == 1

def test_unauthorized_update():
    post_service = PostService(mock_repository)
    with pytest.raises(UnauthorizedError):
        post_service.update_post(1, post_update, user_id=2)
```

2. Integration Tests

```
def test_get_posts_endpoint():
    response = client.get("/posts/")
    assert response.status_code == 200
    assert len(response.json()) > 0

def test_create_post_endpoint():
    response = client.post(
        "/posts/",
        json={
            "title": "Test",
            "content": "Test Content"
        },
        headers={"Authorization": f"Bearer {token}"}
    )
    assert response.status_code == 201
```

Error Handling

1. Domain Errors

```
class PostNotFoundError(Exception):
    """Raised when post is not found"""
    pass

class UnauthorizedError(Exception):
    """Raised when user is not authorized"""
    pass
```

2. HTTP Errors

```
@router.put("/{id}")
def update_post(id: int, post: PostCreate):
    try:
        return use_cases.update_post(id, post, current_user.id)
```

```
    except UnauthorizedError:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Not authorized to update this post"
        )
```

Best Practices

1. Security

- Verify user ownership
- Validate input data
- Prevent SQL injection
- Handle permissions

2. Performance

- Implement pagination
- Optimize queries
- Cache responses
- Index frequently queried fields

3. Maintainability

- Follow clean architecture
- Document APIs
- Write tests
- Handle errors gracefully