Votes Module Documentation

Overview

The votes module implements a comprehensive voting system that allows users to interact with posts through upvotes. Key responsibilities include:

1. Vote management (creation and deletion)
2. Vote counting and statistics
3. User vote tracking
4. Vote-based post ranking
5. Integration with post visibility

Detailed Component Analysis

1. Vote Repository Interface

Location: `votes/domain/repositories/vote_repository.py`

The `VoteRepository` defines the contract for vote data operations, ensuring consistent vote handling across the application.

```
class VoteRepository(ABC):
    @abstractmethod
    def create(self, vote: VoteCreate, user_id: int) -> Vote:
        """
        Creates a new vote record.

        Implementation requirements:
        1. Validate vote uniqueness
        2. Set creation timestamp
        3. Link to user and post
        4. Update vote counts

        Args:
            vote (VoteCreate): Vote data (post_id, direction)
            user_id (int): ID of voting user

        Returns:
            Vote: Created vote record

        Used by:
            - VoteService for vote creation
            - Direct usage in voting endpoint
        """
        pass

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

```
        """
        Checks for existing vote.

        Implementation details:
        1. Composite key lookup
        2. Efficient query plan
        3. Cache integration

        Used by:
            - VoteService for duplicate check
            - Vote validation
            - Vote status check
        """
        pass

    @abstractmethod
    def delete(self, post_id: int, user_id: int) -> bool:
        """
        Removes a vote record.

        Implementation requirements:
        1. Verify vote existence
        2. Handle cascading updates
        3. Update vote counts

        Returns:
            bool: True if vote was deleted
        """
        pass

    @abstractmethod
    def get_vote_count(self, post_id: int) -> int:
        """
        Gets total votes for a post.

        Performance optimizations:
        1. Counter cache usage
        2. Efficient counting query
        3. Result caching

        Used by:
            - Post display
            - Sorting algorithms
            - Trending calculations
        """
        pass
```

2. Vote Model Implementation

Location: `votes/infrastructure/models.py`

The SQLAlchemy model defines the database structure for votes:

```python
class VoteModel(Base):
    """
    Database representation of a vote.

    Table structure:
    - id: Primary key
    - user_id: Foreign key to users
    - post_id: Foreign key to posts
    - created_at: Vote timestamp

    Constraints:
    1. Unique constraint (user_id, post_id)
    2. Foreign key constraints
    3. Cascade delete rules

    Indexes:
    1. Primary key (id)
    2. Composite (user_id, post_id)
    3. Created_at (for analytics)
    """
    __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()')
    )

    __table_args__ = (
        UniqueConstraint('user_id', 'post_id', name='unique_user_post_vote'),
    )
```

3. Vote Service Implementation

Location: `votes/domain/services/vote_service.py`

The service layer contains core voting logic and business rules:

```python
class VoteService:
    """
    Core business logic for vote management.

    Responsibilities:
    1. Vote validation
    2. Vote state management
    3. Vote counting
    4. Integration with posts
    """

    def __init__(self, vote_repository: VoteRepository, post_service:
PostService):
        """
        Initialize with required dependencies.

        Dependencies:
        1. VoteRepository: For vote operations
        2. PostService: For post updates
        """
        self.vote_repository = vote_repository
        self.post_service = post_service

    def vote_post(self, vote: VoteCreate, current_user: User) -> Vote:
        """
        Handles complete voting process.

        Business rules:
        1. One vote per user per post
        2. Can remove own vote
        3. Cannot vote on own posts
        4. Rate limiting applies

        Implementation steps:
        1. Verify post exists
        2. Check previous votes
        3. Apply vote direction
        4. Update post statistics

        Error cases:
        1. Post not found
        2. Already voted
        3. Self-voting
        4. Rate limited
        """
        # Verify post exists
        post = self.post_service.get_post(vote.post_id)
        if not post:
            raise HTTPException(
                status_code=404,
                detail="Post not found"
            )
```

```
        # Prevent self-voting
        if post.owner_id == current_user.id:
            raise HTTPException(
                status_code=400,
                detail="Cannot vote on your own post"
            )

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

        if vote.dir == 1:  # Upvote
            if existing_vote:
                raise HTTPException(
                    status_code=409,
                    detail="Already voted on this post"
                )
            new_vote = self.vote_repository.create(vote, current_user.id)
        else:  # Remove vote
            if not existing_vote:
                raise HTTPException(
                    status_code=404,
                    detail="Vote not found"
                )
            self.vote_repository.delete(vote.post_id, current_user.id)
            new_vote = None

        # Update post vote count
        self._update_post_vote_count(vote.post_id)

        return new_vote

    def _update_post_vote_count(self, post_id: int):
        """
        Updates post vote count.

        Implementation:
        1. Get current count
        2. Update post record
        3. Handle cache invalidation
        """
        vote_count = self.vote_repository.get_vote_count(post_id)
        self.post_service.update_post_votes(post_id, vote_count)
```

4. Vote Use Cases

Location: `votes/application/vote_use_cases.py`

Orchestrates the application flow for voting operations:

```python
class VoteUseCases:
    """
    Application use cases for vote management.

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

    def __init__(self, vote_service: VoteService):
        self.vote_service = vote_service

    async def vote_post(self, vote: VoteCreate, current_user: User) -> Vote:
        """
        Complete voting flow.

        Steps:
        1. Validate request
        2. Apply vote
        3. Handle notifications
        4. Update rankings

        Error handling:
        - Invalid vote data
        - Permission denied
        - Rate limiting
        - Duplicate votes
        """
        try:
            return self.vote_service.vote_post(vote, current_user)
        except HTTPException:
            raise
        except Exception as e:
            raise HTTPException(
                status_code=500,
                detail="Vote operation failed"
            )
```

Advanced Features

1. Vote Analytics

```python
def get_vote_analytics(self, post_id: int) -> VoteAnalytics:
    """
    Comprehensive vote analysis.

    Metrics:
    1. Total vote count
```

```
    2. Vote velocity
    3. User demographics
    4. Time patterns

    Uses:
    1. Trending detection
    2. Content ranking
    3. User engagement
    """
    return self.vote_repository.get_analytics(post_id)
```

2. Rate Limiting

```
def check_vote_rate_limit(self, user_id: int) -> bool:
    """
    Vote rate limiting implementation.

    Rules:
    1. Max votes per minute
    2. Max votes per hour
    3. Account age requirements
    4. Reputation requirements
    """
    return self.rate_limiter.check_limit(
        user_id,
        "vote",
        VOTE_RATE_LIMIT
    )
```

Integration Points

1. Post System Integration

- Vote count updates
- Post ranking
- Content visibility

2. User System Integration

- Vote permissions
- User reputation
- Activity tracking

3. Analytics System

- Vote patterns
- User engagement
- Content popularity

Performance Optimizations

1. **Database Optimization**:
    - Efficient indexes
    - Denormalized counts
    - Batch updates
2. **Caching Strategy**:
    - Vote counts
    - User vote status
    - Popular content
3. **Query Optimization**:
    - Composite keys
    - Covering indexes
    - Query planning

Security Considerations

1. **Vote Manipulation Prevention**:
    - Rate limiting
    - IP tracking
    - Suspicious pattern detection
2. **Data Integrity**:
    - Transaction management
    - Constraint enforcement
    - Audit logging
3. **Access Control**:
    - User authentication
    - Permission validation
    - Vote ownership

Layer Implementation

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

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

```
# votes/domain/entities/vote.py
from pydantic import BaseModel, conint
from datetime import datetime

class VoteBase(BaseModel):
```

```
    """Base vote attributes"""
    post_id: int
    dir: conint(le=1, ge=0)  # 1 = upvote, 0 = downvote

class VoteCreate(VoteBase):
    """Schema for vote creation"""
    pass

class Vote(VoteBase):
    """Schema for vote responses"""
    id: int
    user_id: int
    created_at: datetime

    class Config:
        from_attributes = True
```

These entities:

- Define vote data structures
- Validate vote direction
- Handle vote metadata
- Link votes to posts and users

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

```
# votes/domain/repositories/vote_repository.py
from abc import ABC, abstractmethod
from typing import List, Optional
from ..entities.vote import Vote, VoteCreate

class VoteRepository(ABC):
    """Abstract interface for vote data access"""

    @abstractmethod
    def create(self, vote: VoteCreate, user_id: int) -> Vote:
        """Create a new vote"""
        pass

    @abstractmethod
    def get_by_post_and_user(
        self,
        post_id: int,
        user_id: int
    ) -> Optional[Vote]:
        """Get vote by post and user"""
        pass

    @abstractmethod
    def delete(self, post_id: int, user_id: int) -> None:
        """Delete vote by post and user"""
```

```
        pass

    @abstractmethod
    def get_post_votes(self, post_id: int) -> int:
        """Get total votes for a post"""
        pass

    @abstractmethod
    def get_user_votes(self, user_id: int) -> List[Vote]:
        """Get all votes by user"""
        pass
```

The repository interface:

- Defines vote operations
- Handles vote queries
- Manages vote counts
- Links votes to users

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

```
# votes/domain/services/vote_service.py
from typing import Optional
from ..repositories.vote_repository import VoteRepository
from ..entities.vote import Vote, VoteCreate
from ...domain.exceptions import VoteError
from ...posts.domain.services.post_service import PostService

class VoteService:
    """Core vote business logic"""

    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, user_id: int) -> Optional[Vote]:
        """Vote on a post"""
        # Verify post exists
        self.post_service.get_post(vote.post_id)

        # Check existing vote
        existing_vote = self.vote_repository.get_by_post_and_user(
            vote.post_id,
            user_id
        )

        if vote.dir == 1:
```

```
            if existing_vote:
                raise VoteError("Already voted on this post")
            return self.vote_repository.create(vote, user_id)
        else:
            if not existing_vote:
                raise VoteError("Vote does not exist")
            self.vote_repository.delete(vote.post_id, user_id)
            return None

    def get_vote_count(self, post_id: int) -> int:
        """Get total votes for a post"""
        return self.vote_repository.get_post_votes(post_id)
```

The vote service:

- Implements voting logic
- Handles vote validation
- Manages vote state
- Integrates with posts

2. Application Layer (`votes/application/`)

2.1 Use Cases (`votes/application/vote_use_cases.py`)

```
# votes/application/vote_use_cases.py
from fastapi import HTTPException, status
from ..domain.entities.vote import Vote, VoteCreate
from ..domain.services.vote_service import VoteService
from ..domain.exceptions import VoteError
from ...posts.domain.exceptions import PostNotFoundError

class VoteUseCases:
    """Application use cases for votes"""

    def __init__(self, vote_service: VoteService):
        self.vote_service = vote_service

    def vote_post(self, vote: VoteCreate, user_id: int) -> Optional[Vote]:
        """Vote on post use case"""
        try:
            return self.vote_service.vote_post(vote, user_id)
        except PostNotFoundError as e:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail=str(e)
            )
        except VoteError as e:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail=str(e)
            )
```

```
            except Exception as e:
                raise HTTPException(
                    status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
                    detail=str(e)
                )

        def get_vote_count(self, post_id: int) -> int:
            """Get post vote count use case"""
            try:
                return self.vote_service.get_vote_count(post_id)
            except PostNotFoundError as e:
                raise HTTPException(
                    status_code=status.HTTP_404_NOT_FOUND,
                    detail=str(e)
                )
```

The use cases:

- Orchestrate voting flows
- Handle exceptions
- Map to HTTP errors
- Manage transactions

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

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

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

class VoteModel(Base):
    """SQLAlchemy model for votes table"""

    __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()')
    )

    # Relationships
    user = relationship("UserModel")
    post = relationship("PostModel", back_populates="votes")

    # Constraints
    __table_args__ = (
        UniqueConstraint('user_id', 'post_id', name='unique_user_post_vote'),
    )
```

The database model:

- Maps vote data to database
- Defines relationships
- Enforces constraints
- Manages timestamps

3.2 Repository Implementation (`votes/infrastructure/vote_repository_impl.py`)

```
# votes/infrastructure/vote_repository_impl.py
from sqlalchemy.orm import Session
from sqlalchemy import func
from typing import List, Optional
from ..domain.repositories.vote_repository import VoteRepository
from ..domain.entities.vote import Vote, VoteCreate
from .models import VoteModel

class SQLAlchemyVoteRepository(VoteRepository):
    """SQLAlchemy implementation of VoteRepository"""

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

    def create(self, vote: VoteCreate, user_id: int) -> Vote:
        """Create new vote in database"""
        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_by_post_and_user(
        self,
        post_id: int,
```

```
        user_id: int
    ) -> Optional[Vote]:
        """Get vote by post and user from database"""
        vote = self.db.query(VoteModel)\
            .filter(
                VoteModel.post_id == post_id,
                VoteModel.user_id == user_id
            )\
            .first()
        return Vote.from_orm(vote) if vote else None

    def delete(self, post_id: int, user_id: int) -> None:
        """Delete vote from database"""
        self.db.query(VoteModel)\
            .filter(
                VoteModel.post_id == post_id,
                VoteModel.user_id == user_id
            )\
            .delete()
        self.db.commit()

    def get_post_votes(self, post_id: int) -> int:
        """Get total votes for post from database"""
        return self.db.query(func.count(VoteModel.id))\
            .filter(VoteModel.post_id == post_id)\
            .scalar()

    def get_user_votes(self, user_id: int) -> List[Vote]:
        """Get all user votes from database"""
        votes = self.db.query(VoteModel)\
            .filter(VoteModel.user_id == user_id)\
            .all()
        return [Vote.from_orm(v) for v in votes]
```

The repository implementation:

- Implements repository interface
- Handles database operations
- Manages vote queries
- Ensures data integrity

4. Presentation Layer (`votes/presentation/`)

4.1 API Routes (`votes/presentation/router.py`)

```
# votes/presentation/router.py
from fastapi import APIRouter, Depends, status
from sqlalchemy.orm import Session
from typing import Optional
from ...shared.infrastructure.database import get_db
from ...auth.domain.entities.user import User
```

```python
from ...auth.presentation.oauth2 import get_current_user
from ..domain.entities.vote import Vote, VoteCreate
from ..application.vote_use_cases import VoteUseCases
from ..domain.services.vote_service import VoteService
from ..infrastructure.vote_repository_impl import SQLAlchemyVoteRepository
from ...posts.domain.services.post_service import PostService
from ...posts.infrastructure.post_repository_impl import SQLAlchemyPostRepository

router = APIRouter(
    prefix="/vote",
    tags=["Vote"]
)

@router.post("/", status_code=status.HTTP_201_CREATED)
def vote(
    vote: VoteCreate,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user)
) -> Optional[Vote]:
    """Vote on post endpoint"""
    post_service = PostService(
        post_repository=SQLAlchemyPostRepository(db)
    )
    vote_service = VoteService(
        vote_repository=SQLAlchemyVoteRepository(db),
        post_service=post_service
    )
    use_cases = VoteUseCases(vote_service)
    return use_cases.vote_post(vote, current_user.id)

@router.get("/{post_id}/count")
def get_vote_count(
    post_id: int,
    db: Session = Depends(get_db)
) -> int:
    """Get post vote count endpoint"""
    post_service = PostService(
        post_repository=SQLAlchemyPostRepository(db)
    )
    vote_service = VoteService(
        vote_repository=SQLAlchemyVoteRepository(db),
        post_service=post_service
    )
    use_cases = VoteUseCases(vote_service)
    return use_cases.get_vote_count(post_id)
```

The API routes:

- Define vote endpoints
- Handle authentication
- Inject dependencies
- Return vote status

Vote Operations Flow

1. Vote Creation Flow

1. Client sends POST to `/vote/` with vote data
2. Auth middleware validates token
3. Router validates vote direction
4. Use case checks existing vote
5. Repository creates/deletes vote
6. Router returns vote status

2. Vote Count Flow

1. Client sends GET to `/vote/{post_id}/count`
2. Router validates post ID
3. Use case retrieves count
4. Repository calculates total
5. Router returns count

Integration Points

1. Auth Module Integration

- User authentication
- Vote ownership
- User activity tracking

2. Posts Module Integration

- Post existence validation
- Vote count aggregation
- Post statistics

Testing

1. Unit Tests

```
def test_vote_creation():
    vote_service = VoteService(mock_vote_repo, mock_post_service)
    vote = VoteCreate(post_id=1, dir=1)
    result = vote_service.vote_post(vote, user_id=1)
    assert result.post_id == vote.post_id

def test_duplicate_vote():
    vote_service = VoteService(mock_vote_repo, mock_post_service)
    with pytest.raises(VoteError):
        vote_service.vote_post(vote, user_id=1)
```

2. Integration Tests

```
def test_vote_endpoint():
    response = client.post(
        "/vote/",
        json={"post_id": 1, "dir": 1},
        headers={"Authorization": f"Bearer {token}"}
    )
    assert response.status_code == 201

def test_vote_count_endpoint():
    response = client.get("/vote/1/count")
    assert response.status_code == 200
    assert isinstance(response.json(), int)
```

Error Handling

1. Domain Errors

```
class VoteError(Exception):
    """Base class for vote-related errors"""
    pass

class DuplicateVoteError(VoteError):
    """Raised when user tries to vote twice"""
    pass

class InvalidVoteError(VoteError):
    """Raised when vote direction is invalid"""
    pass
```

2. HTTP Errors

```
@router.post("/")
def vote(vote: VoteCreate):
    try:
        return use_cases.vote_post(vote, current_user.id)
    except DuplicateVoteError:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Already voted on this post"
        )
```

Best Practices

1. Data Integrity

- Enforce unique votes

- Handle race conditions
- Maintain referential integrity
- Validate vote direction

2. Performance

- Index vote queries
- Cache vote counts
- Optimize aggregations
- Handle concurrent votes

3. Security

- Verify user authentication
- Validate post existence
- Prevent vote manipulation
- Log voting activity

4. Maintainability

- Follow clean architecture
- Document vote logic
- Write comprehensive tests
- Handle edge cases