Authentication Module Documentation

Overview

The authentication module is a critical component that handles all aspects of user identity and security in the application. It's responsible for:

1. User registration and account creation
2. Secure login and session management
3. Password encryption and verification
4. Token-based authentication using JWT
5. Protected route access control

Detailed Component Analysis

1. User Repository Interface

Location: `auth/domain/repositories/user_repository.py`

The `UserRepository` is an abstract interface that defines the contract for user data operations. It's crucial for maintaining the dependency inversion principle of clean architecture.

```
class UserRepository(ABC):
    @abstractmethod
    def create(self, user: UserCreate) -> User:
        """
        Creates a new user in the system.

        Args:
            user (UserCreate): Contains validated user data (email, username,
password)

        Returns:
            User: Created user entity with generated ID

        Used by:
            - AuthService during user registration
            - Direct usage in registration endpoint
        """
        pass

    @abstractmethod
    def get_by_id(self, user_id: int) -> Optional[User]:
        """
        Retrieves user by their unique ID.

        Args:
            user_id (int): Unique identifier of the user
```

```
        Returns:
            Optional[User]: User if found, None otherwise

        Used by:
            - AuthService for token verification
            - PostService for ownership verification
            - VoteService for vote validation
        """
        pass

    @abstractmethod
    def get_by_email(self, email: str) -> Optional[User]:
        """
        Finds user by email address.

        Args:
            email (str): User's email address

        Returns:
            Optional[User]: User if found, None otherwise

        Used by:
            - AuthService during login
            - Registration to check email uniqueness
        """
        pass
```

Why it's required:

1. **Abstraction**: Separates interface from implementation, allowing different storage solutions
2. **Testing**: Enables easy mocking for unit tests
3. **Flexibility**: New implementations can be added without changing business logic

2. Auth Service Implementation

Location: `auth/domain/services/auth_service.py`

The `AuthService` contains core authentication logic and security features. It's the heart of the authentication system.

```
class AuthService:
    def __init__(self, user_repository: UserRepository):
        """
        Initialize auth service with required dependencies.

        Args:
            user_repository: Interface to user data operations
        """
        self.user_repository = user_repository
        self.pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
```

```
    def verify_password(self, plain_password: str, hashed_password: str) -> bool:
        """
        Securely verifies a password against its hash.

        Implementation:
        1. Uses bcrypt for comparison (timing-attack safe)
        2. Handles both plain and hashed versions

        Used in:
        - Login process
        - Password change verification
        """
        return self.pwd_context.verify(plain_password, hashed_password)

    def create_access_token(self, data: dict) -> str:
        """
        Generates a JWT token with user data.

        Implementation details:
        1. Uses JWT standard with HS256 algorithm
        2. Includes expiration time (30 minutes by default)
        3. Signs token with application secret key

        Token structure:
        {
            "sub": "user_id",
            "exp": expiration_timestamp,
            "iat": issued_at_timestamp,
            ...additional_claims
        }

        Security features:
        1. Expiration time prevents token reuse
        2. Signature verification prevents tampering
        3. Payload encryption for sensitive data

        Used in:
        - Login endpoint to generate session token
        - Token refresh operations
        """
        to_encode = data.copy()
        expire = datetime.utcnow() +
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
        to_encode.update({
            "exp": expire,
            "iat": datetime.utcnow()
        })
        return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
```

3. User Repository Implementation

Location: `auth/infrastructure/user_repository_impl.py`

The SQLAlchemy implementation of the user repository provides concrete data access logic:

```
class SQLAlchemyUserRepository(UserRepository):
    def __init__(self, db: Session, auth_service: AuthService):
        """
        Initialize with database session and auth service.

        Implementation details:
        1. Uses SQLAlchemy ORM for database operations
        2. Maintains transaction integrity
        3. Handles connection pooling

        Args:
            db (Session): SQLAlchemy database session
            auth_service: For password hashing
        """
        self.db = db
        self.auth_service = auth_service

    def create(self, user: UserCreate) -> User:
        """
        Creates new user with proper password hashing.

        Implementation steps:
        1. Hash password using bcrypt
        2. Create UserModel instance
        3. Add to database session
        4. Commit transaction
        5. Refresh to get generated ID
        6. Convert to domain entity

        Security features:
        1. Password never stored in plain text
        2. Unique constraints on email/username
        3. SQL injection protection via ORM

        Error handling:
        1. Unique constraint violations
        2. Database connection issues
        3. Invalid data types
        """
        hashed_password = self.auth_service.get_password_hash(user.password)
        db_user = UserModel(
            email=user.email,
            username=user.username,
            password=hashed_password
        )
        try:
            self.db.add(db_user)
            self.db.commit()
            self.db.refresh(db_user)
            return User.from_orm(db_user)
        except IntegrityError:
```

```
            self.db.rollback()
            raise HTTPException(
                status_code=400,
                detail="Email or username already registered"
            )
```

4. Authentication Flow

Registration Process:

1. **Request Validation**:
    - Validates email format
    - Checks password strength
    - Verifies username requirements
2. **User Creation**:

```
@router.post("/register", response_model=User)
async def register(user: UserCreate, db: Session = Depends(get_db)):
    """
    Complete registration flow:
    1. Validate input data
    2. Check for existing users
    3. Create user record
    4. Generate welcome email
    5. Return user data

    Error handling:
    - Duplicate email/username
    - Invalid data formats
    - Database errors
    """
    auth_use_cases = get_auth_use_cases(db)
    return await auth_use_cases.register(user)
```

Login Process:

1. **Credential Verification**:

```
@router.post("/login", response_model=Token)
async def login(
    form_data: OAuth2PasswordRequestForm = Depends(),
    db: Session = Depends(get_db)
):
    """
    Secure login implementation:
    1. Validate credentials
```

```
        2. Generate JWT token
        3. Set cookie headers
        4. Return token response

        Security features:
        1. Rate limiting
        2. Brute force protection
        3. Secure cookie settings
        """
        auth_use_cases = get_auth_use_cases(db)
        token = await auth_use_cases.login(
            form_data.username,
            form_data.password
        )
        response = JSONResponse(content=token.dict())
        response.set_cookie(
            key="access_token",
            value=f"Bearer {token.access_token}",
            httponly=True,
            secure=True,
            samesite='lax'
        )
        return response
```

5. Protected Routes Implementation

The `get_current_user` dependency protects routes requiring authentication:

```
async def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: Session = Depends(get_db)
) -> User:
    """
    Validates JWT token and returns current user.

    Security checks:
    1. Token presence
    2. Token format validation
    3. Signature verification
    4. Expiration check
    5. User existence verification

    Error responses:
    - 401 Unauthorized: Invalid/expired token
    - 403 Forbidden: Insufficient permissions
    - 404 Not Found: User doesn't exist

    Usage:
    @router.get("/protected")
    async def protected_route(user: User = Depends(get_current_user)):
        return {"message": f"Hello {user.username}"}
```

```
    """
    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)
```

Security Considerations

1. **Password Security**:
    - Bcrypt hashing with salt
    - Minimum password requirements
    - Password history tracking
2. **Token Security**:
    - Short expiration time
    - Secure cookie settings
    - CSRF protection
3. **Request Security**:
    - Rate limiting
    - Input validation
    - SQL injection protection
4. **Session Management**:
    - Secure session handling
    - Token revocation
    - Session timeout

Integration Points

1. **Posts Module**:
    - User ownership verification
    - Author information
    - Permission checks
2. **Votes Module**:

- User vote tracking
- Permission validation
- Vote uniqueness

3. **API Security**:

- Route protection
- Permission middleware
- Error handling

Layer Implementation

2.1 Domain Layer (`auth/domain/`)

2.1.1 Entities (`auth/domain/entities/`)

```
# auth/domain/entities/user.py
from pydantic import BaseModel, EmailStr
from datetime import datetime

class UserBase(BaseModel):
    """Base user attributes shared by all user-related schemas"""
    email: EmailStr
    username: str

class UserCreate(UserBase):
    """Schema for user registration, extends base with password"""
    password: str

class User(UserBase):
    """Schema for user responses, adds system fields"""
    id: int
    created_at: datetime

    class Config:
        from_attributes = True
```

These entities:

- Define core user data structures
- Handle data validation via Pydantic
- Are used across all layers
- Remain independent of persistence details

2.1.2 Repository Interface (`auth/domain/repositories/`)

```
# auth/domain/repositories/user_repository.py
from abc import ABC, abstractmethod
from typing import Optional
```

```python
from ..entities.user import User, UserCreate

class UserRepository(ABC):
    """Abstract interface for user data access"""

    @abstractmethod
    def create(self, user: UserCreate) -> User:
        """Create a new user"""
        pass

    @abstractmethod
    def get_by_id(self, user_id: int) -> Optional[User]:
        """Get user by ID"""
        pass

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

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

The repository interface:

- Defines data access contract
- Uses domain entities
- Is implementation-agnostic
- Enforces consistent data access patterns

2.1.3 Domain Services (`auth/domain/services/`)

```python
# auth/domain/services/auth_service.py
from datetime import datetime, timedelta
from jose import jwt
from passlib.context import CryptContext
from ..repositories.user_repository import UserRepository

class AuthService:
    """Core authentication business logic"""

    def __init__(self, user_repository: UserRepository):
        self.user_repository = user_repository
        self.pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

    def verify_password(self, plain_password: str, hashed_password: str) -> bool:
        """Verify password against hash"""
        return self.pwd_context.verify(plain_password, hashed_password)
```

```
    def get_password_hash(self, password: str) -> str:
        """Generate password hash"""
        return self.pwd_context.hash(password)

    def create_access_token(self, data: dict) -> str:
        """Create JWT access token"""
        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)

    def verify_token(self, token: str) -> dict:
        """Verify and decode JWT token"""
        try:
            payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
            return payload
        except JWTError:
            raise InvalidTokenError()
```

The auth service:

- Implements core authentication logic
- Handles password hashing/verification
- Manages JWT token creation/verification
- Uses repository for data access

2.2 Application Layer (`auth/application/`)

2.2.1 Use Cases (`auth/application/auth_use_cases.py`)

```
# auth/application/auth_use_cases.py
from fastapi import HTTPException, status
from ..domain.entities.user import User, UserCreate
from ..domain.services.auth_service import AuthService
from ..domain.repositories.user_repository import UserRepository

class AuthUseCases:
    """Application use cases for authentication"""

    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:
        """Register a new user"""
        # Check if email exists
        if self.user_repository.get_by_email(user.email):
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
```

```
                detail="Email already registered"
            )

        # Check if username exists
        if self.user_repository.get_by_username(user.username):
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Username already taken"
            )

        return self.user_repository.create(user)

    def authenticate_user(self, username: str, password: str) -> User:
        """Authenticate user credentials"""
        user = self.user_repository.get_by_username(username)
        if not user:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Invalid credentials"
            )

        if not self.auth_service.verify_password(password, user.password):
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Invalid credentials"
            )

        return user

    def create_token(self, user: User) -> str:
        """Create access token for user"""
        token_data = {
            "sub": user.username,
            "email": user.email,
            "id": user.id
        }
        return self.auth_service.create_access_token(token_data)
```

The use cases:

- Orchestrate domain objects
- Implement application flows
- Handle business validation
- Map domain exceptions to HTTP

2.3 Infrastructure Layer (`auth/infrastructure/`)

2.3.1 Database Models (`auth/infrastructure/models.py`)

```
# auth/infrastructure/models.py
from sqlalchemy import Column, Integer, String, DateTime
```

```
from sqlalchemy.sql.expression import text
from ...shared.infrastructure.database import Base

class UserModel(Base):
    """SQLAlchemy model for users table"""

    __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()')
    )
```

The database model:

- Maps user data to database
- Defines schema constraints
- Handles relationships
- Manages timestamps

2.3.2 Repository Implementation (`auth/infrastructure/user_repository_impl.py`)

```
# auth/infrastructure/user_repository_impl.py
from sqlalchemy.orm import Session
from ..domain.repositories.user_repository import UserRepository
from ..domain.entities.user import User, UserCreate
from ..domain.services.auth_service import AuthService
from .models import UserModel

class SQLAlchemyUserRepository(UserRepository):
    """SQLAlchemy implementation of UserRepository"""

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

    def create(self, user: UserCreate) -> User:
        """Create new user in database"""
        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)

    def get_by_id(self, user_id: int) -> Optional[User]:
        """Get user by ID from database"""
        user = self.db.query(UserModel)\
            .filter(UserModel.id == user_id)\
            .first()
        return User.from_orm(user) if user else None

    def get_by_email(self, email: str) -> Optional[User]:
        """Get user by email from database"""
        user = self.db.query(UserModel)\
            .filter(UserModel.email == email)\
            .first()
        return User.from_orm(user) if user else None

    def get_by_username(self, username: str) -> Optional[User]:
        """Get user by username from database"""
        user = self.db.query(UserModel)\
            .filter(UserModel.username == username)\
            .first()
        return User.from_orm(user) if user else None
```

The repository implementation:

- Implements repository interface
- Handles database operations
- Converts between models and entities
- Manages transactions

2.4 Presentation Layer (`auth/presentation/`)

2.4.1 API Routes (`auth/presentation/router.py`)

```
# auth/presentation/router.py
from fastapi import APIRouter, Depends, status
from fastapi.security import OAuth2PasswordRequestForm
from sqlalchemy.orm import Session
from typing import Annotated
from ...shared.infrastructure.database import get_db
from ..domain.entities.user import User, UserCreate
from ..application.auth_use_cases import AuthUseCases
from ..domain.services.auth_service import AuthService
from ..infrastructure.user_repository_impl import SQLAlchemyUserRepository

router = APIRouter(
    prefix="/auth",
```

```
    tags=["Authentication"]
)

@router.post("/register", response_model=User)
def register(
    user: UserCreate,
    db: Session = Depends(get_db)
):
    """Register new user endpoint"""
    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)

@router.post("/login")
def login(
    form_data: Annotated[OAuth2PasswordRequestForm, Depends()],
    db: Session = Depends(get_db)
):
    """Login endpoint"""
    auth_service = AuthService(
        user_repository=SQLAlchemyUserRepository(db)
    )
    use_cases = AuthUseCases(
        user_repository=SQLAlchemyUserRepository(db, auth_service),
        auth_service=auth_service
    )

    user = use_cases.authenticate_user(
        form_data.username,
        form_data.password
    )
    access_token = use_cases.create_token(user)

    return {
        "access_token": access_token,
        "token_type": "bearer"
    }

@router.get("/me", response_model=User)
def get_current_user(
    current_user: Annotated[User, Depends(get_current_user)]
):
    """Get current user endpoint"""
    return current_user
```

The API routes:

- Define HTTP endpoints

- Handle request/response
- Manage authentication
- Inject dependencies

3. Authentication Flow

3.1 Registration Flow

1. Client sends POST to `/auth/register` with user data
2. Router validates request data using `UserCreate` schema
3. Use case checks for existing email/username
4. Auth service hashes password
5. Repository saves user to database
6. Router returns user data using `User` schema

3.2 Login Flow

1. Client sends POST to `/auth/login` with credentials
2. Router validates form data
3. Use case attempts authentication
4. Auth service verifies password
5. Use case generates JWT token
6. Router returns token response

3.3 Protected Route Flow

1. Client sends request with Bearer token
2. Auth middleware extracts token
3. Auth service verifies token
4. Repository loads user data
5. Request proceeds with user context

4. Security Features

4.1 Password Security

- Passwords hashed using bcrypt
- Salt automatically managed
- Configurable work factor
- Secure comparison

4.2 JWT Implementation

- Signed using HS256
- Contains user claims
- Configurable expiration
- Refresh token support

4.3 Protection Against

- Brute force attacks
- Token replay
- SQL injection
- Password exposure

5. Integration Points

5.1 Posts Module

- User ownership of posts
- Author information
- Access control

5.2 Votes Module

- User voting records
- Vote authorization
- User activity tracking

6. Configuration

6.1 Environment Variables

```
SECRET_KEY=your-secret-key
ALGORITHM=HS256
ACCESS_TOKEN_EXPIRE_MINUTES=30
```

6.2 Dependencies

```
from fastapi.security import OAuth2PasswordBearer
from passlib.context import CryptContext

# Password hashing
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# OAuth2 scheme
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="auth/login")
```

7. Testing

7.1 Unit Tests

```
def test_password_hashing():
    auth_service = AuthService(mock_repository)
    password = "test123"
    hashed = auth_service.get_password_hash(password)
```

```
    assert auth_service.verify_password(password, hashed)

def test_user_registration():
    use_cases = AuthUseCases(mock_repository, mock_auth_service)
    user = UserCreate(
        email="test@example.com",
        username="testuser",
        password="test123"
    )
    result = use_cases.register_user(user)
    assert result.email == user.email
```

7.2 Integration Tests

```
def test_login_endpoint():
    response = client.post(
        "/auth/login",
        data={
            "username": "testuser",
            "password": "test123"
        }
    )
    assert response.status_code == 200
    assert "access_token" in response.json()
```

8. Error Handling

8.1 Domain Errors

```
class InvalidCredentialsError(Exception):
    """Raised when credentials are invalid"""
    pass

class UserExistsError(Exception):
    """Raised when user already exists"""
    pass
```

8.2 HTTP Errors

```
@router.post("/login")
def login(credentials: LoginCredentials):
    try:
        user = use_cases.authenticate_user(
            credentials.username,
            credentials.password
        )
```

```
    except InvalidCredentialsError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid credentials"
        )
```

9. Best Practices

9.1 Security

- Never store plain passwords
- Use secure token generation
- Implement rate limiting
- Log security events

9.2 Performance

- Index username and email
- Cache user sessions
- Optimize token validation
- Use connection pooling

9.3 Maintainability

- Follow clean architecture
- Document security decisions
- Use type hints
- Write comprehensive tests