FastAPI Sections 8-10: Authentication, Relationships & Voting

Section 8: Authentication & Users

1. Creating Users Table

sql

```sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR NOT NULL UNIQUE,
    password VARCHAR NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

2. User Registration

- Create user registration endpoint
- Validate email format and uniqueness
- Hash password before storing
- Return user data (exclude password)

3. Password Hashing

python

```python
from passlib.context import CryptContext

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

def hash_password(password: str):
    return pwd_context.hash(password)

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

4. FastAPI Routers

- Organize endpoints using APIRouter
- Separate authentication logic into dedicated router
- Use router prefixes (`/users`, `/auth`)

- Add router tags for documentation

5. JWT Token Authentication

Key Concepts:

- JWT = JSON Web Token
- Contains: Header, Payload, Signature
- Stateless authentication
- Expires after set time

Implementation:

```python
from jose import JWTError, jwt
from datetime import datetime, timedelta

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

def create_access_token(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)
```

6. Login Process

1. User provides email/password
2. Verify credentials against database
3. Create JWT token if valid
4. Return token to client
5. Client includes token in Authorization header

7. OAuth2 PasswordRequestForm

python

```python
from fastapi.security import OAuth2PasswordRequestForm

@app.post("/login")
def login(user_credentials: OAuth2PasswordRequestForm = Depends()):
    # user_credentials.username (email)
    # user_credentials.password
```

8. Protecting Routes

python

```python
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="login")

def get_current_user(token: str = Depends(oauth2_scheme)):
    # Verify and decode token
    # Return user data
```

Section 9: Relationships

1. SQL Relationship Basics

- **One-to-Many**: One user can have many posts
- **Many-to-Many**: Posts can have many votes, users can vote on many posts
- **Foreign Keys**: Link tables together

2. Postgres Foreign Keys

sql

```sql
ALTER TABLE posts
ADD COLUMN owner_id INTEGER NOT NULL;

ALTER TABLE posts
ADD CONSTRAINT posts_users_fk
FOREIGN KEY (owner_id) REFERENCES users(id)
ON DELETE CASCADE;
```

3. SQLAlchemy Foreign Keys

```python
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship

class Post(Base):
    __tablename__ = "posts"
    
    id = Column(Integer, primary_key=True)
    title = Column(String, nullable=False)
    content = Column(String, nullable=False)
    owner_id = Column(Integer, ForeignKey("users.id"), nullable=False)
    
    # Relationship
    owner = relationship("User")
```

4. Update Post Schema

```python
from pydantic import BaseModel

class PostResponse(BaseModel):
    id: int
    title: str
    content: str
    owner_id: int
    owner: UserResponse  # Include user data
    
    class Config:
        from_attributes = True
```

5. Ownership-Based Operations

- **Create**: Automatically assign `owner_id` from current user
- **Delete**: Only allow deletion of own posts
- **Update**: Only allow updates to own posts
- **Read**: Option to filter by current user's posts

python

```python
@app.delete("/posts/{id}")
def delete_post(id: int, current_user: int = Depends(get_current_user)):
    post = db.query(Post).filter(Post.id == id).first()

    if post.owner_id != current_user.id:
        raise HTTPException(status_code=403, detail="Not authorized")

    db.delete(post)
    db.commit()
```

6. SQLAlchemy Relationships

python

```python
class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    email = Column(String, nullable=False, unique=True)

    # Back reference to posts
    posts = relationship("Post", back_populates="owner")

class Post(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True)
    owner_id = Column(Integer, ForeignKey("users.id"))

    # Forward reference to user
    owner = relationship("User", back_populates="posts")
```

7. Query Parameters

python

```python
@app.get("/posts")
def get_posts(
    db: Session = Depends(get_database),
    current_user: int = Depends(get_current_user),
    limit: int = 10,
    skip: int = 0,
    search: Optional[str] = ""
):
    posts = db.query(Post).filter(
        Post.title.contains(search)
    ).limit(limit).offset(skip).all()

    return posts
```

8. Environment Variables

python

```python
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    database_hostname: str
    database_port: str
    database_password: str
    database_name: str
    database_username: str
    secret_key: str
    algorithm: str
    access_token_expire_minutes: int

    class Config:
        env_file = ".env"

settings = Settings()
```

Section 10: Vote/Like System

1. Vote/Like Theory

- Users can vote on posts (like/unlike)
- Each user can only vote once per post

- Votes are stored in separate table
- Common social media pattern

2. Votes Table Structure

sql

```sql
CREATE TABLE votes (
    user_id INTEGER NOT NULL,
    post_id INTEGER NOT NULL,
    PRIMARY KEY (user_id, post_id),
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
    FOREIGN KEY (post_id) REFERENCES posts(id) ON DELETE CASCADE
);
```

3. Votes SQLAlchemy Model

python

```python
class Vote(Base):
    __tablename__ = "votes"
    
    user_id = Column(Integer, ForeignKey("users.id"), primary_key=True)
    post_id = Column(Integer, ForeignKey("posts.id"), primary_key=True)
```

4. Votes Route Implementation

python

```python
from pydantic import BaseModel

class Vote(BaseModel):
    post_id: int
    dir: int  # 1 for upvote, 0 for removing vote

@app.post("/vote")
def vote(vote: Vote, current_user: int = Depends(get_current_user)):
    # Check if post exists
    post = db.query(Post).filter(Post.id == vote.post_id).first()
    if not post:
        raise HTTPException(status_code=404, detail="Post not found")
    
    # Check if vote already exists
    vote_query = db.query(Vote).filter(
        Vote.post_id == vote.post_id,
        Vote.user_id == current_user.id
    )
    found_vote = vote_query.first()
    
    if vote.dir == 1:  # Upvote
        if found_vote:
            raise HTTPException(status_code=409, detail="Already voted")
        
        new_vote = Vote(post_id=vote.post_id, user_id=current_user.id)
        db.add(new_vote)
        db.commit()
        return {"message": "Successfully added vote"}
    
    else:  # Remove vote
        if not found_vote:
            raise HTTPException(status_code=404, detail="Vote not found")
        
        vote_query.delete(synchronize_session=False)
        db.commit()
        return {"message": "Successfully removed vote"}
```

5. SQL Joins

- **INNER JOIN**: Only matching records
- **LEFT JOIN**: All records from left table

- **RIGHT JOIN**: All records from right table
- **FULL OUTER JOIN**: All records from both tables

6. Joins in SQLAlchemy

python

```python
from sqlalchemy import func

# Get posts with vote counts
results = db.query(Post, func.count(Vote.post_id).label("votes")).join(
    Vote, Vote.post_id == Post.id, isouter=True
).group_by(Post.id).all()

# Access results
for post, votes in results:
    print(f"Post: {post.title}, Votes: {votes}")
```

7. Enhanced Post Response with Votes

python

```python
class PostOut(BaseModel):
    Post: Post
    votes: int

    class Config:
        from_attributes = True

@app.get("/posts", response_model=List[PostOut])
def get_posts():
    results = db.query(Post, func.count(Vote.post_id).label("votes")).join(
        Vote, Vote.post_id == Post.id, isouter=True
    ).group_by(Post.id).all()

    return results
```

Key Takeaways

Security Best Practices

- Always hash passwords before storing
- Use strong secret keys for JWT

- Implement proper token expiration
- Validate user ownership for operations
- Use environment variables for sensitive data

Database Design

- Use foreign keys to maintain referential integrity
- Implement proper relationships between tables
- Use composite primary keys for many-to-many relationships
- Consider CASCADE options for foreign keys

API Design

- Use proper HTTP status codes
- Implement consistent error handling
- Use query parameters for filtering and pagination
- Structure responses with clear data models
- Organize endpoints with routers and prefixes