🚀 FastAPI Project Structure (Enhanced Version)

Based on your Node.js backend with descriptive naming conventions, here's the enhanced FastAPI project structure:

📁 Enhanced Project Structure

```
fastapi-backend/
├── .env.sample
├── .gitignore
├── requirements.txt
├── pyproject.toml
├── README.md
├── main.py
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── config.py
│   ├── constants.py
│   ├── api/
│   │   ├── __init__.py
│   │   ├── dependencies.py
│   │   └── v1/
│   │       ├── __init__.py
│   │       ├── api.py
│   │       └── routes/
│   │           ├── __init__.py
│   │           ├── auth.route.py
│   │           ├── user.route.py
│   │           ├── video.route.py
│   │           ├── comment.route.py
│   │           ├── like.route.py
│   │           ├── playlist.route.py
│   │           ├── subscription.route.py
│   │           └── tweet.route.py
│   ├── core/
│   │   ├── __init__.py
│   │   ├── config.py
│   │   ├── security.py
│   │   └── settings.py
│   ├── db/
│   │   ├── __init__.py
│   │   ├── base.py
│   │   ├── database.py
│   │   ├── session.py
│   │   └── migrations/
│   │       └── __init__.py
```

```
│   ├── models/
│   │   ├── __init__.py
│   │   ├── base.model.py
│   │   ├── user.model.py
│   │   ├── video.model.py
│   │   ├── comment.model.py
│   │   ├── like.model.py
│   │   ├── playlist.model.py
│   │   ├── subscription.model.py
│   │   └── tweet.model.py
│   ├── schemas/
│   │   ├── __init__.py
│   │   ├── user.schema.py
│   │   ├── video.schema.py
│   │   ├── comment.schema.py
│   │   ├── like.schema.py
│   │   ├── playlist.schema.py
│   │   ├── subscription.schema.py
│   │   └── tweet.schema.py
│   ├── controllers/
│   │   ├── __init__.py
│   │   ├── user.controller.py
│   │   ├── video.controller.py
│   │   ├── comment.controller.py
│   │   ├── like.controller.py
│   │   ├── playlist.controller.py
│   │   ├── subscription.controller.py
│   │   └── tweet.controller.py
│   ├── services/
│   │   ├── __init__.py
│   │   ├── user.service.py
│   │   ├── video.service.py
│   │   ├── comment.service.py
│   │   ├── like.service.py
│   │   ├── playlist.service.py
│   │   ├── subscription.service.py
│   │   ├── tweet.service.py
│   │   └── auth.service.py
│   ├── middlewares/
│   │   ├── __init__.py
│   │   ├── cors.middleware.py
│   │   ├── auth.middleware.py
│   │   ├── logging.middleware.py
│   │   └── error.middleware.py
│   └── utils/
│       ├── __init__.py
│       ├── helpers.util.py
│       ├── validators.util.py
│       ├── exceptions.util.py
│       ├── responses.util.py
│       └── constants.util.py
```

```
├── tests/
│   ├── __init__.py
│   ├── conftest.py
│   ├── api/
│   │   ├── __init__.py
│   │   ├── test_user.route.py
│   │   ├── test_video.route.py
│   │   └── test_auth.route.py
│   ├── controllers/
│   │   ├── __init__.py
│   │   ├── test_user.controller.py
│   │   └── test_video.controller.py
│   └── services/
│       ├── __init__.py
│       ├── test_user.service.py
│       └── test_video.service.py
└── static/
    ├── css/
    ├── js/
    ├── images/
    └── uploads/
        ├── videos/
        ├── thumbnails/
        └── avatars/
```

🎯 Enhanced File Examples

app/routes/user.route.py

```python
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from app.db.session import get_db
from app.controllers.user.controller import UserController
from app.schemas.user.schema import UserCreate, UserResponse, UserUpdate
from app.middlewares.auth.middleware import get_current_user

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

@router.post("/", response_model=UserResponse, status_code=status.HTTP_201_CREATED)
async def create_user(user: UserCreate, db: Session = Depends(get_db)):
    """Create a new user account"""
    return await UserController.create_user(db, user)

@router.get("/me", response_model=UserResponse)
async def get_current_user_profile(current_user = Depends(get_current_user)):
    """Get current user profile"""
    return current_user

@router.get("/{user_id}", response_model=UserResponse)
```

```python
async def get_user_by_id(user_id: int, db: Session = Depends(get_db)):
    """Get user by ID"""
    return await UserController.get_user_by_id(db, user_id)

@router.patch("/me", response_model=UserResponse)
async def update_user_profile(
    user_update: UserUpdate,
    current_user = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """Update current user profile"""
    return await UserController.update_user(db, current_user.id, user_update)

@router.delete("/me", status_code=status.HTTP_204_NO_CONTENT)
async def delete_user_account(
    current_user = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """Delete current user account"""
    await UserController.delete_user(db, current_user.id)
```

app/models/user.model.py

```python
from sqlalchemy import Column, Integer, String, DateTime, Boolean, Text
from sqlalchemy.sql import func
from sqlalchemy.orm import relationship
from app.db.base import Base

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    username = Column(String(50), unique=True, index=True, nullable=False)
    email = Column(String(100), unique=True, index=True, nullable=False)
    full_name = Column(String(100), nullable=True)
    avatar = Column(String(255), nullable=True)
    cover_image = Column(String(255), nullable=True)
    watch_history = Column(Text, nullable=True)
    password = Column(String(255), nullable=False)
    refresh_token = Column(Text, nullable=True)
    is_active = Column(Boolean, default=True)
    is_verified = Column(Boolean, default=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())

    # Relationships
    videos = relationship("Video", back_populates="owner")
    comments = relationship("Comment", back_populates="owner")
    likes = relationship("Like", back_populates="user")
```

```
    playlists = relationship("Playlist", back_populates="owner")
    subscriptions_given = relationship("Subscription",
foreign_keys="Subscription.subscriber_id", back_populates="subscriber")
    subscriptions_received = relationship("Subscription",
foreign_keys="Subscription.channel_id", back_populates="channel")
    tweets = relationship("Tweet", back_populates="owner")
```

app/models/video.model.py

```
from sqlalchemy import Column, Integer, String, DateTime, Boolean, Text, ForeignKey
from sqlalchemy.sql import func
from sqlalchemy.orm import relationship
from app.db.base import Base

class Video(Base):
    __tablename__ = "videos"

    id = Column(Integer, primary_key=True, index=True)
    video_file = Column(String(255), nullable=False)
    thumbnail = Column(String(255), nullable=False)
    title = Column(String(200), nullable=False)
    description = Column(Text, nullable=True)
    duration = Column(Integer, nullable=False)  # in seconds
    views = Column(Integer, default=0)
    is_published = Column(Boolean, default=False)
    owner_id = Column(Integer, ForeignKey("users.id"), nullable=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())

    # Relationships
    owner = relationship("User", back_populates="videos")
    comments = relationship("Comment", back_populates="video")
    likes = relationship("Like", back_populates="video")
```

app/controllers/user.controller.py

```
from sqlalchemy.orm import Session
from fastapi import HTTPException, status
from app.models.user.model import User
from app.schemas.user.schema import UserCreate, UserUpdate
from app.services.user.service import UserService
from app.utils.responses.util import success_response, error_response

class UserController:
    @staticmethod
    async def create_user(db: Session, user: UserCreate):
```

```
        """Create a new user"""
        try:
            # Check if user already exists
            existing_user = await UserService.get_user_by_email(db, user.email)
            if existing_user:
                raise HTTPException(
                    status_code=status.HTTP_400_BAD_REQUEST,
                    detail="User with this email already exists"
                )

            new_user = await UserService.create_user(db, user)
            return success_response(
                data=new_user,
                message="User created successfully"
            )
        except Exception as e:
            raise HTTPException(
                status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
                detail=str(e)
            )

    @staticmethod
    async def get_user_by_id(db: Session, user_id: int):
        """Get user by ID"""
        user = await UserService.get_user_by_id(db, user_id)
        if not user:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="User not found"
            )
        return success_response(data=user)

    @staticmethod
    async def update_user(db: Session, user_id: int, user_update: UserUpdate):
        """Update user"""
        updated_user = await UserService.update_user(db, user_id, user_update)
        if not updated_user:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="User not found"
            )
        return success_response(
            data=updated_user,
            message="User updated successfully"
        )
```

app/services/user.service.py

```python
from sqlalchemy.orm import Session
from sqlalchemy import or_
from app.models.user.model import User
from app.schemas.user.schema import UserCreate, UserUpdate
from app.utils.helpers.util import hash_password
from typing import Optional

class UserService:
    @staticmethod
    async def create_user(db: Session, user: UserCreate) -> User:
        """Create a new user"""
        hashed_password = hash_password(user.password)
        db_user = User(
            username=user.username,
            email=user.email,
            full_name=user.full_name,
            password=hashed_password
        )
        db.add(db_user)
        db.commit()
        db.refresh(db_user)
        return db_user

    @staticmethod
    async def get_user_by_id(db: Session, user_id: int) -> Optional[User]:
        """Get user by ID"""
        return db.query(User).filter(User.id == user_id).first()

    @staticmethod
    async def get_user_by_email(db: Session, email: str) -> Optional[User]:
        """Get user by email"""
        return db.query(User).filter(User.email == email).first()

    @staticmethod
    async def get_user_by_username(db: Session, username: str) -> Optional[User]:
        """Get user by username"""
        return db.query(User).filter(User.username == username).first()

    @staticmethod
    async def update_user(db: Session, user_id: int, user_update: UserUpdate) ->
Optional[User]:
        """Update user"""
        db_user = db.query(User).filter(User.id == user_id).first()
        if not db_user:
            return None

        update_data = user_update.dict(exclude_unset=True)
        for field, value in update_data.items():
            setattr(db_user, field, value)
```

```
        db.commit()
        db.refresh(db_user)
        return db_user
```

app/schemas/user.schema.py

```
from pydantic import BaseModel, EmailStr, validator
from datetime import datetime
from typing import Optional, List

class UserBase(BaseModel):
    username: str
    email: EmailStr
    full_name: Optional[str] = None

class UserCreate(UserBase):
    password: str

    @validator('password')
    def validate_password(cls, v):
        if len(v) < 8:
            raise ValueError('Password must be at least 8 characters long')
        return v

class UserUpdate(BaseModel):
    username: Optional[str] = None
    email: Optional[EmailStr] = None
    full_name: Optional[str] = None
    avatar: Optional[str] = None
    cover_image: Optional[str] = None

class UserResponse(UserBase):
    id: int
    avatar: Optional[str] = None
    cover_image: Optional[str] = None
    is_active: bool
    is_verified: bool
    created_at: datetime
    updated_at: Optional[datetime] = None

    class Config:
        from_attributes = True

class UserLogin(BaseModel):
    username: str  # Can be username or email
    password: str
```

🎨 Enhanced Naming Conventions

Component	Convention	Example
Routes	`{resource}.route.py`	`user.route.py`, `video.route.py`
Models	`{resource}.model.py`	`user.model.py`, `comment.model.py`
Controllers	`{resource}.controller.py`	`user.controller.py`, `playlist.controller.py`
Services	`{resource}.service.py`	`user.service.py`, `subscription.service.py`
Schemas	`{resource}.schema.py`	`user.schema.py`, `tweet.schema.py`
Middlewares	`{purpose}.middleware.py`	`auth.middleware.py`, `cors.middleware.py`
Utils	`{purpose}.util.py`	`helpers.util.py`, `validators.util.py`
Tests	`test_{resource}.{type}.py`	`test_user.route.py`, `test_video.service.py`

🌟 Key Enhancements

- 🎯 **Clear Purpose**: Each file name immediately indicates its function
- 📱 **Social Media Ready**: Complete structure for YouTube/Twitter-like platform
- 🔄 **Consistent**: All naming follows the same `.{type}.py` pattern
- 🧪 **Test Ready**: Organized test structure matching main codebase
- 📁 **Media Support**: Static folders for uploads (videos, thumbnails, avatars)
- 🔒 **Security First**: Dedicated auth middleware and services
- 📊 **Response Standards**: Consistent response utilities
- 🚀 **Scalable**: Easy to add new features following the same pattern

---

🏗️ FastAPI Module-Functionality Structure

📖 Overview

The **Module-Functionality** approach organizes files based on feature domains rather than file types. Each module contains all the components it needs (routes, models, schemas, services) in one cohesive package.

> 💡 **Best For**: Monolithic applications with multiple domains and complex business logic

🎯 Key Benefits

Benefit	Description
🎯 **Domain Focus**	All related functionality grouped together
🚀 **Development Efficiency**	Faster feature development and debugging
🔍 **Better Maintainability**	Easy to locate and modify domain-specific code

Benefit	Description
📦 **Scalability**	Simple to add new modules without affecting others
👥 **Team Collaboration**	Teams can work on different modules independently

📁 Complete Project Structure

```
fastapi-project/
├── 📁 alembic/                    # Database migrations
│   ├── 📄 env.py
│   ├── 📄 script.py.mako
│   └── 📁 versions/
├── 📁 src/                        # Main application source
│   ├── 📁 auth/                   # 🔐 Authentication Module
│   │   ├── 📄 __init__.py
│   │   ├── 📄 router.py           # Auth endpoints & routes
│   │   ├── 📄 schemas.py          # Pydantic request/response models
│   │   ├── 📄 models.py           # Database models (User, Token, etc.)
│   │   ├── 📄 service.py          # Business logic (login, register, etc.)
│   │   ├── 📄 dependencies.py     # Auth dependencies & middleware
│   │   ├── 📄 config.py           # Auth-specific configurations
│   │   ├── 📄 constants.py        # Auth constants & error codes
│   │   ├── 📄 exceptions.py       # Auth-specific exceptions
│   │   └── 📄 utils.py            # Helper functions (password hashing, etc.)
│   ├── 📁 users/                  # 👤 User Management Module
│   │   ├── 📄 __init__.py
│   │   ├── 📄 router.py           # User CRUD endpoints
│   │   ├── 📄 schemas.py          # User request/response schemas
│   │   ├── 📄 models.py           # User database models
│   │   ├── 📄 service.py          # User business logic
│   │   ├── 📄 dependencies.py     # User-specific dependencies
│   │   ├── 📄 constants.py        # User constants
│   │   ├── 📄 exceptions.py       # User-specific exceptions
│   │   └── 📄 utils.py            # User utility functions
│   ├── 📁 posts/                  # 📝 Posts Management Module
│   │   ├── 📄 __init__.py
│   │   ├── 📄 router.py           # Post CRUD endpoints
│   │   ├── 📄 schemas.py          # Post schemas
│   │   ├── 📄 models.py           # Post database models
│   │   ├── 📄 service.py          # Post business logic
│   │   ├── 📄 dependencies.py     # Post dependencies
│   │   ├── 📄 constants.py        # Post constants
│   │   ├── 📄 exceptions.py       # Post exceptions
│   │   └── 📄 utils.py            # Post utilities
│   ├── 📁 aws/                    # ☁️ AWS Integration Module
│   │   ├── 📄 __init__.py
│   │   ├── 📄 client.py           # AWS service clients
│   │   ├── 📄 schemas.py          # AWS request/response schemas
│   │   ├── 📄 config.py           # AWS configurations
```

```
│   │   ├── constants.py        # AWS constants
│   │   ├── exceptions.py       # AWS-specific exceptions
│   │   └── utils.py            # AWS utility functions
│   ├── notifications/          # 🔔 Notification Module
│   │   ├── __init__.py
│   │   ├── router.py
│   │   ├── schemas.py
│   │   ├── models.py
│   │   ├── service.py
│   │   ├── dependencies.py
│   │   ├── constants.py
│   │   ├── exceptions.py
│   │   └── utils.py
│   ├── main.py                 # 🚀 FastAPI app initialization
│   ├── config.py               # 🔧 Global configurations
│   ├── database.py             # 🗄️ Database connection setup
│   ├── models.py               # 🌐 Global database models
│   ├── exceptions.py           # 🚨 Global exception handlers
│   ├── pagination.py           # 📄 Global pagination utilities
│   └── dependencies.py         # 🔗 Global dependencies
├── tests/                      # 🧪 Test Suite
│   ├── __init__.py
│   ├── conftest.py
│   ├── auth/
│   │   ├── __init__.py
│   │   ├── test_router.py
│   │   ├── test_service.py
│   │   └── test_utils.py
│   ├── users/
│   │   ├── __init__.py
│   │   ├── test_router.py
│   │   └── test_service.py
│   ├── posts/
│   │   ├── __init__.py
│   │   ├── test_router.py
│   │   └── test_service.py
│   └── aws/
│       ├── __init__.py
│       └── test_client.py
├── templates/                  # 🎨 Jinja2 templates
│   ├── base.html
│   ├── index.html
│   └── login.html
├── static/                     # 🎭 Static files
│   ├── css/
│   ├── js/
│   └── images/
├── requirements/               # 📦 Dependencies
│   ├── base.txt                # Base requirements
│   ├── dev.txt                 # Development requirements
│   └── prod.txt                # Production requirements
```

```
├── 📄 .env                    # 🔐 Environment variables
├── 📄 .env.example            # 📝 Environment template
├── 📄 .gitignore              # 🚫 Git ignore rules
├── 📄 logging.ini             # 📊 Logging configuration
├── 📄 alembic.ini             # 🗄️ Alembic configuration
├── 📄 docker-compose.yml      # 🐬 Docker configuration
├── 📄 Dockerfile              # 🐬 Docker image definition
└── 📄 README.md               # 📖 Project documentation
```

🧬 Module Architecture

Each module follows a consistent internal structure:

📊 Module Component Breakdown

File	Purpose	Example Content
`router.py`	🛣️ API endpoints and route definitions	`@router.post("/login")`
`schemas.py`	📋 Pydantic models for request/response	`class UserCreate(BaseModel)`
`models.py`	🗄️ SQLAlchemy database models	`class User(Base)`
`service.py`	💼 Business logic and core functionality	`def authenticate_user()`
`dependencies.py`	🔗 FastAPI dependencies and injections	`def get_current_user()`
`config.py`	⚙️ Module-specific configurations	`JWT_SECRET_KEY`
`constants.py`	📝 Constants and error codes	`MIN_PASSWORD_LENGTH = 8`
`exceptions.py`	⚠️ Custom exception classes	`class UserNotFound(HTTPException)`
`utils.py`	🛠️ Helper and utility functions	`def hash_password()`

📝 Implementation Examples

📄 src/auth/router.py

```python
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from src.database import get_db
from src.auth import service, schemas, dependencies

router = APIRouter(prefix="/auth", tags=["authentication"])

@router.post("/login", response_model=schemas.TokenResponse)
async def login(
    credentials: schemas.LoginRequest,
    db: Session = Depends(get_db)
```

```
):
    """User login endpoint"""
    user = await service.authenticate_user(db, credentials.username,
credentials.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid credentials"
        )

    token = service.create_access_token(user.id)
    return schemas.TokenResponse(access_token=token, token_type="bearer")

@router.post("/register", response_model=schemas.UserResponse)
async def register(
    user_data: schemas.UserCreate,
    db: Session = Depends(get_db)
):
    """User registration endpoint"""
    return await service.create_user(db, user_data)

@router.get("/me", response_model=schemas.UserResponse)
async def get_current_user_info(
    current_user = Depends(dependencies.get_current_user)
):
    """Get current user information"""
    return current_user
```

src/auth/service.py

```
from sqlalchemy.orm import Session
from passlib.context import CryptContext
from jose import jwt
from datetime import datetime, timedelta
from src.auth import models, schemas, constants, exceptions
from src.auth.config import AUTH_CONFIG

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

class AuthService:
    @staticmethod
    async def authenticate_user(db: Session, username: str, password: str):
        """Authenticate user with username and password"""
        user = db.query(models.User).filter(
            models.User.username == username
        ).first()

        if not user or not pwd_context.verify(password, user.hashed_password):
            return None
```

```
        return user

    @staticmethod
    def create_access_token(user_id: int) -> str:
        """Create JWT access token"""
        expire = datetime.utcnow() + timedelta(
            minutes=AUTH_CONFIG.ACCESS_TOKEN_EXPIRE_MINUTES
        )

        to_encode = {
            "sub": str(user_id),
            "exp": expire,
            "type": "access"
        }

        return jwt.encode(
            to_encode,
            AUTH_CONFIG.SECRET_KEY,
            algorithm=AUTH_CONFIG.ALGORITHM
        )

    @staticmethod
    async def create_user(db: Session, user_data: schemas.UserCreate):
        """Create new user account"""
        # Check if user exists
        existing_user = db.query(models.User).filter(
            models.User.email == user_data.email
        ).first()

        if existing_user:
            raise exceptions.UserAlreadyExists("User with this email already
exists")

        # Hash password
        hashed_password = pwd_context.hash(user_data.password)

        # Create user
        db_user = models.User(
            username=user_data.username,
            email=user_data.email,
            hashed_password=hashed_password
        )

        db.add(db_user)
        db.commit()
        db.refresh(db_user)

        return db_user
```

src/auth/schemas.py

```python
from pydantic import BaseModel, EmailStr, validator
from datetime import datetime
from typing import Optional

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

class UserCreate(UserBase):
    password: str

    @validator('password')
    def validate_password(cls, v):
        if len(v) < 8:
            raise ValueError('Password must be at least 8 characters')
        return v

class UserResponse(UserBase):
    id: int
    is_active: bool
    created_at: datetime

    class Config:
        from_attributes = True

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

class TokenResponse(BaseModel):
    access_token: str
    token_type: str = "bearer"
```

src/auth/constants.py

```python
# Authentication Constants
MIN_PASSWORD_LENGTH = 8
MAX_PASSWORD_LENGTH = 128
USERNAME_MIN_LENGTH = 3
USERNAME_MAX_LENGTH = 50

# Error Codes
class ErrorCodes:
    INVALID_CREDENTIALS = "AUTH_001"
    USER_NOT_FOUND = "AUTH_002"
    USER_ALREADY_EXISTS = "AUTH_003"
    INVALID_TOKEN = "AUTH_004"
```

```
    TOKEN_EXPIRED = "AUTH_005"
    INSUFFICIENT_PERMISSIONS = "AUTH_006"

# Token Types
class TokenTypes:
    ACCESS = "access"
    REFRESH = "refresh"
    RESET_PASSWORD = "reset_password"
```

src/auth/exceptions.py

```
from fastapi import HTTPException, status
from src.auth.constants import ErrorCodes

class AuthException(HTTPException):
    """Base authentication exception"""
    pass

class UserNotFound(AuthException):
    def __init__(self, detail: str = "User not found"):
        super().__init__(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=detail,
            headers={"error-code": ErrorCodes.USER_NOT_FOUND}
        )

class UserAlreadyExists(AuthException):
    def __init__(self, detail: str = "User already exists"):
        super().__init__(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=detail,
            headers={"error-code": ErrorCodes.USER_ALREADY_EXISTS}
        )

class InvalidCredentials(AuthException):
    def __init__(self, detail: str = "Invalid credentials"):
        super().__init__(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail=detail,
            headers={"error-code": ErrorCodes.INVALID_CREDENTIALS}
        )

class InvalidToken(AuthException):
    def __init__(self, detail: str = "Invalid token"):
        super().__init__(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail=detail,
```

```
            headers={"error-code": ErrorCodes.INVALID_TOKEN}
        )
```

🔗 Cross-Module Imports

When modules need to interact with each other, use explicit imports:

```
# ✅ Good: Explicit module imports
from src.auth import constants as auth_constants
from src.notifications import service as notification_service
from src.posts.constants import ErrorCode as PostErrorCode
from src.users.models import User

# ❌ Avoid: Ambiguous imports
from constants import ErrorCode  # Which module's constants?
from service import send_email   # Which service?
```

📄 Example Cross-Module Usage

```
# src/posts/service.py
from src.auth import service as auth_service
from src.notifications import service as notification_service
from src.users import models as user_models

class PostService:
    @staticmethod
    async def create_post(db: Session, post_data: schemas.PostCreate, current_user):
        # Create the post
        new_post = models.Post(**post_data.dict(), author_id=current_user.id)
        db.add(new_post)
        db.commit()

        # Send notification to followers
        followers = await user_models.get_user_followers(db, current_user.id)
        await notification_service.notify_new_post(followers, new_post)

        return new_post
```

📊 Project Configuration

📄 src/main.py

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
```

```python
from src.auth.router import router as auth_router
from src.users.router import router as users_router
from src.posts.router import router as posts_router
from src.config import settings

app = FastAPI(
    title="Module-Based FastAPI Project",
    description="FastAPI project organized by module functionality",
    version="1.0.0"
)

# Middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=settings.ALLOWED_HOSTS,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Include routers
app.include_router(auth_router)
app.include_router(users_router)
app.include_router(posts_router)

@app.get("/")
async def root():
    return {"message": "Welcome to Module-Based FastAPI Project"}
```

🌟 Advantages of Module-Functionality Structure

✅ Benefits:

Advantage	Description
🎯 **Domain Cohesion**	All related code lives together
🚀 **Faster Development**	Easy to locate and modify features
👥 **Team Scalability**	Multiple developers can work on different modules
🔧 **Easy Testing**	Module-specific test isolation
📦 **Microservice Ready**	Easy to extract modules into separate services
🔄 **Code Reusability**	Modules can be shared across projects

⚠️ Considerations:

Challenge	Solution
Code Duplication	Use global utilities and shared modules
Cross-Module Dependencies	Use explicit imports and dependency injection
Module Boundaries	Clear domain separation and well-defined interfaces

🎯 When to Use This Structure

✅ Perfect For:

- 🏢 **Large Applications** with multiple business domains
- 👥 **Team Development** with domain-specific teams
- 🔄 **Microservice Migration** planning
- 📈 **Scalable Applications** that will grow significantly

❌ Might Be Overkill For:

- 🔸 Simple APIs with few endpoints
- 🔸 Prototypes or proof-of-concepts
- 🔸 Single-developer projects
- 🔸 Applications with minimal business logic

🚀 Getting Started

1. Create Module Template

```
mkdir -p src/{auth,users,posts}
for module in auth users posts; do
    touch
src/$module/{__init__.py,router.py,schemas.py,models.py,service.py,dependencies.py,c
onfig.py,constants.py,exceptions.py,utils.py}
done
```

2. Set Up Dependencies

```
pip install fastapi uvicorn sqlalchemy alembic psycopg2-binary
pip install python-jose[cryptography] passlib[bcrypt] python-multipart
```

3. Initialize Database

```
alembic init alembic
alembic revision --autogenerate -m "Initial migration"
```

```
alembic upgrade head
```
