Complete FastAPI Models Guide: Best Practices & User Management System

📋 Table of Contents

🎯 Understanding Pydantic BaseModel

What is BaseModel?

`BaseModel` is the foundation class from Pydantic that provides:

- **Data validation** ✅
- **Serialization/Deserialization** 🔄
- **Type checking** 📝
- **Automatic documentation** 📚

```
from pydantic import BaseModel

class ExampleModel(BaseModel):
    name: str
    age: int
    email: str
```

When to Inherit from BaseModel?

Always inherit from BaseModel when:

- Creating request schemas (input validation)
- Creating response schemas (output formatting)
- Creating internal data models
- Working with FastAPI route parameters

🏗️ Model Architecture & Inheritance

The 3-Layer Model Pattern

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Request       │──▶│   Internal      │──▶│   Response      │
│   Models        │    │   Models        │    │   Models        │
│                 │    │                 │    │                 │
│ • Validation    │    │ • DB Operations │    │ • API Output    │
│ • Input Schema  │    │ • Full Data     │    │ • Safe Data     │
└─────────────────┘    └─────────────────┘    └─────────────────┘
```

📁 Project Structure

```
app/
├── models/
│   ├── __init__.py
│   ├── base.py              # Base models
│   ├── request/
│   │   ├── __init__.py
│   │   ├── auth.py          # Auth request models
│   │   └── user.py          # User request models
│   ├── response/
│   │   ├── __init__.py
│   │   ├── auth.py          # Auth response models
│   │   └── user.py          # User response models
│   └── internal/
│       ├── __init__.py
│       └── user.py          # Internal user models
├── routers/
│   ├── auth.py
│   └── user.py
├── services/
│   ├── auth.py
│   └── user.py
└── main.py
```

🧱 Core Model Types

1. Base Models

```python
# models/base.py
from datetime import datetime
from typing import Optional
from pydantic import BaseModel, Field
from bson import ObjectId

class PyObjectId(ObjectId):
    @classmethod
```

```
    def __get_validators__(cls):
        yield cls.validate

    @classmethod
    def validate(cls, v):
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid objectid")
        return ObjectId(v)

    @classmethod
    def __modify_schema__(cls, field_schema):
        field_schema.update(type="string")

class BaseDBModel(BaseModel):
    """Base model for database entities"""
    id: Optional[PyObjectId] = Field(default_factory=PyObjectId, alias="_id")
    created_at: datetime = Field(default_factory=datetime.utcnow)
    updated_at: datetime = Field(default_factory=datetime.utcnow)

    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}

class BaseResponse(BaseModel):
    """Base response model"""
    success: bool = True
    message: str = "Operation successful"

class ErrorResponse(BaseModel):
    """Error response model"""
    success: bool = False
    message: str
    error_code: Optional[str] = None
```

2. Request Models

```
# models/request/auth.py
from pydantic import BaseModel, EmailStr, Field, validator
from typing import Optional
import re

class RegisterUserRequest(BaseModel):
    username: str = Field(..., min_length=3, max_length=20)
    email: EmailStr
    password: str = Field(..., min_length=8)
    full_name: str = Field(..., min_length=2, max_length=50)

    @validator('username')
```

```
    def validate_username(cls, v):
        if not re.match(r'^[a-zA-Z0-9_]+$', v):
            raise ValueError('Username can only contain letters, numbers, and
underscores')
        return v

    @validator('password')
    def validate_password(cls, v):
        if not re.search(r'[A-Z]', v):
            raise ValueError('Password must contain at least one uppercase letter')
        if not re.search(r'[a-z]', v):
            raise ValueError('Password must contain at least one lowercase letter')
        if not re.search(r'\d', v):
            raise ValueError('Password must contain at least one digit')
        return v

class LoginUserRequest(BaseModel):
    email: EmailStr
    password: str

class RefreshTokenRequest(BaseModel):
    refresh_token: str

class ChangePasswordRequest(BaseModel):
    current_password: str
    new_password: str = Field(..., min_length=8)

    @validator('new_password')
    def validate_new_password(cls, v):
        if not re.search(r'[A-Z]', v):
            raise ValueError('Password must contain at least one uppercase letter')
        if not re.search(r'[a-z]', v):
            raise ValueError('Password must contain at least one lowercase letter')
        if not re.search(r'\d', v):
            raise ValueError('Password must contain at least one digit')
        return v
```

```
# models/request/user.py
from pydantic import BaseModel, Field
from typing import Optional

class UpdateAccountDetailsRequest(BaseModel):
    full_name: Optional[str] = Field(None, min_length=2, max_length=50)
    bio: Optional[str] = Field(None, max_length=500)
    location: Optional[str] = Field(None, max_length=100)
    website: Optional[str] = Field(None, max_length=200)

class UpdateUserAvatarRequest(BaseModel):
    avatar_url: str = Field(..., description="URL of the uploaded avatar image")
```

```python
class UpdateUserCoverImageRequest(BaseModel):
    cover_image_url: str = Field(..., description="URL of the uploaded cover image")
```

3. Internal Models

```python
# models/internal/user.py
from datetime import datetime
from typing import Optional, List, Dict, Any
from pydantic import EmailStr, Field
from models.base import BaseDBModel

class UserInDB(BaseDBModel):
    """Complete user model for database operations"""
    username: str = Field(..., unique=True, index=True)
    email: EmailStr = Field(..., unique=True, index=True)
    hashed_password: str
    full_name: str
    is_active: bool = True
    is_verified: bool = False
    is_superuser: bool = False

    # Profile information
    bio: Optional[str] = None
    location: Optional[str] = None
    website: Optional[str] = None
    avatar_url: Optional[str] = None
    cover_image_url: Optional[str] = None

    # Timestamps
    last_login: Optional[datetime] = None
    email_verified_at: Optional[datetime] = None

    # Settings
    preferences: Dict[str, Any] = Field(default_factory=dict)
    privacy_settings: Dict[str, bool] = Field(default_factory=lambda: {
        "show_email": False,
        "show_location": True,
        "show_website": True
    })

    # Social features
    followers_count: int = 0
    following_count: int = 0
    posts_count: int = 0

    class Config:
        collection_name = "users"
        schema_extra = {
```

```
            "example": {
                "username": "johndoe",
                "email": "john@example.com",
                "full_name": "John Doe",
                "bio": "Software developer passionate about Python",
                "location": "San Francisco, CA"
            }
        }

class UserSession(BaseDBModel):
    """User session model for managing authentication"""
    user_id: str
    access_token: str
    refresh_token: str
    device_info: Optional[str] = None
    ip_address: Optional[str] = None
    expires_at: datetime
    is_active: bool = True

    class Config:
        collection_name = "user_sessions"
```

4. Response Models

```
# models/response/auth.py
from datetime import datetime
from typing import Optional
from pydantic import EmailStr
from models.base import BaseResponse

class TokenResponse(BaseResponse):
    access_token: str
    refresh_token: str
    token_type: str = "bearer"
    expires_in: int  # seconds

class LoginResponse(TokenResponse):
    user: "UserResponse"

class RegisterResponse(BaseResponse):
    user: "UserResponse"
    message: str = "User registered successfully"

# Forward reference resolution
from models.response.user import UserResponse
LoginResponse.update_forward_refs()
RegisterResponse.update_forward_refs()
```

```python
# models/response/user.py
from datetime import datetime
from typing import Optional, Dict, Any
from pydantic import BaseModel, EmailStr
from models.base import BaseResponse

class UserResponse(BaseModel):
    """Public user information"""
    id: str
    username: str
    email: EmailStr
    full_name: str
    bio: Optional[str] = None
    location: Optional[str] = None
    website: Optional[str] = None
    avatar_url: Optional[str] = None
    cover_image_url: Optional[str] = None
    is_verified: bool
    followers_count: int = 0
    following_count: int = 0
    posts_count: int = 0
    created_at: datetime
    last_login: Optional[datetime] = None

class CurrentUserResponse(UserResponse):
    """Extended user info for the authenticated user"""
    email_verified_at: Optional[datetime] = None
    preferences: Dict[str, Any] = {}
    privacy_settings: Dict[str, bool] = {}

class UpdateUserResponse(BaseResponse):
    user: UserResponse
    message: str = "User updated successfully"

class UserListResponse(BaseResponse):
    users: list[UserResponse]
    total: int
    page: int
    limit: int
```

🔐 Complete User Management Implementation

Authentication Service

```python
# services/auth.py
from datetime import datetime, timedelta
from typing import Optional
```

```python
from fastapi import HTTPException, status
from passlib.context import CryptContext
from jose import JWTError, jwt
from models.internal.user import UserInDB, UserSession
from models.request.auth import RegisterUserRequest, LoginUserRequest

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

class AuthService:
    def __init__(self, secret_key: str, algorithm: str = "HS256"):
        self.secret_key = secret_key
        self.algorithm = algorithm
        self.access_token_expire_minutes = 30
        self.refresh_token_expire_days = 7

    def verify_password(self, plain_password: str, hashed_password: str) -> bool:
        return pwd_context.verify(plain_password, hashed_password)

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

    def create_access_token(self, data: dict) -> str:
        to_encode = data.copy()
        expire = datetime.utcnow() +
timedelta(minutes=self.access_token_expire_minutes)
        to_encode.update({"exp": expire, "type": "access"})
        return jwt.encode(to_encode, self.secret_key, algorithm=self.algorithm)

    def create_refresh_token(self, data: dict) -> str:
        to_encode = data.copy()
        expire = datetime.utcnow() + timedelta(days=self.refresh_token_expire_days)
        to_encode.update({"exp": expire, "type": "refresh"})
        return jwt.encode(to_encode, self.secret_key, algorithm=self.algorithm)

    def verify_token(self, token: str, token_type: str = "access") -> dict:
        try:
            payload = jwt.decode(token, self.secret_key, algorithms=
[self.algorithm])
            if payload.get("type") != token_type:
                raise HTTPException(
                    status_code=status.HTTP_401_UNAUTHORIZED,
                    detail="Invalid token type"
                )
            return payload
        except JWTError:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Could not validate credentials"
            )

    async def register_user(self, user_data: RegisterUserRequest) -> UserInDB:
```

```
        # Check if user exists
        existing_user = await self.get_user_by_email(user_data.email)
        if existing_user:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Email already registered"
            )

        existing_username = await self.get_user_by_username(user_data.username)
        if existing_username:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Username already taken"
            )

        # Create new user
        hashed_password = self.get_password_hash(user_data.password)
        user = UserInDB(
            username=user_data.username,
            email=user_data.email,
            hashed_password=hashed_password,
            full_name=user_data.full_name
        )

        # Save to database (implement based on your DB choice)
        saved_user = await self.save_user(user)
        return saved_user

    async def authenticate_user(self, email: str, password: str) ->
Optional[UserInDB]:
        user = await self.get_user_by_email(email)
        if not user or not self.verify_password(password, user.hashed_password):
            return None

        # Update last login
        user.last_login = datetime.utcnow()
        await self.update_user(user)
        return user
```

User Routes

```
# routers/auth.py
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import HTTPBearer
from models.request.auth import (
    RegisterUserRequest,
    LoginUserRequest,
    RefreshTokenRequest,
    ChangePasswordRequest
```

```python
)
from models.response.auth import LoginResponse, RegisterResponse, TokenResponse
from models.response.user import CurrentUserResponse
from services.auth import AuthService

router = APIRouter(prefix="/auth", tags=["Authentication"])
security = HTTPBearer()

@router.post("/register", response_model=RegisterResponse)
async def register_user(user_data: RegisterUserRequest):
    """Register a new user"""
    auth_service = AuthService(secret_key="your-secret-key")
    user = await auth_service.register_user(user_data)

    return RegisterResponse(
        user=UserResponse.from_orm(user),
        message="User registered successfully"
    )

@router.post("/login", response_model=LoginResponse)
async def login_user(credentials: LoginUserRequest):
    """Authenticate user and return tokens"""
    auth_service = AuthService(secret_key="your-secret-key")
    user = await auth_service.authenticate_user(
        credentials.email,
        credentials.password
    )

    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect email or password"
        )

    access_token = auth_service.create_access_token(
        data={"sub": str(user.id), "email": user.email}
    )
    refresh_token = auth_service.create_refresh_token(
        data={"sub": str(user.id)}
    )

    return LoginResponse(
        access_token=access_token,
        refresh_token=refresh_token,
        expires_in=1800,  # 30 minutes
        user=UserResponse.from_orm(user)
    )

@router.post("/logout")
async def logout_user(token: str = Depends(security)):
    """Logout user and invalidate tokens"""
```

```python
    # Implement token blacklisting logic
    return {"message": "Successfully logged out"}

@router.post("/refresh", response_model=TokenResponse)
async def refresh_access_token(refresh_data: RefreshTokenRequest):
    """Refresh access token using refresh token"""
    auth_service = AuthService(secret_key="your-secret-key")
    payload = auth_service.verify_token(refresh_data.refresh_token, "refresh")

    new_access_token = auth_service.create_access_token(
        data={"sub": payload["sub"]}
    )

    return TokenResponse(
        access_token=new_access_token,
        refresh_token=refresh_data.refresh_token,
        expires_in=1800
    )

@router.put("/change-password")
async def change_current_password(
    password_data: ChangePasswordRequest,
    current_user: UserInDB = Depends(get_current_user)
):
    """Change user's password"""
    auth_service = AuthService(secret_key="your-secret-key")

    if not auth_service.verify_password(
        password_data.current_password,
        current_user.hashed_password
    ):
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Incorrect current password"
        )

    current_user.hashed_password = auth_service.get_password_hash(
        password_data.new_password
    )
    await auth_service.update_user(current_user)

    return {"message": "Password changed successfully"}

@router.get("/me", response_model=CurrentUserResponse)
async def get_current_user(current_user: UserInDB = Depends(get_current_user)):
    """Get current user information"""
    return CurrentUserResponse.from_orm(current_user)
```

User Management Routes

```python
# routers/user.py
from fastapi import APIRouter, Depends, UploadFile, File
from models.request.user import (
    UpdateAccountDetailsRequest,
    UpdateUserAvatarRequest,
    UpdateUserCoverImageRequest
)
from models.response.user import UpdateUserResponse, CurrentUserResponse
from models.internal.user import UserInDB

router = APIRouter(prefix="/user", tags=["User Management"])

@router.put("/update-account", response_model=UpdateUserResponse)
async def update_account_details(
    update_data: UpdateAccountDetailsRequest,
    current_user: UserInDB = Depends(get_current_user)
):
    """Update user account details"""
    update_dict = update_data.dict(exclude_unset=True)

    for field, value in update_dict.items():
        setattr(current_user, field, value)

    current_user.updated_at = datetime.utcnow()
    updated_user = await update_user_in_db(current_user)

    return UpdateUserResponse(
        user=UserResponse.from_orm(updated_user),
        message="Account details updated successfully"
    )

@router.put("/update-avatar", response_model=UpdateUserResponse)
async def update_user_avatar(
    avatar_data: UpdateUserAvatarRequest,
    current_user: UserInDB = Depends(get_current_user)
):
    """Update user avatar"""
    current_user.avatar_url = avatar_data.avatar_url
    current_user.updated_at = datetime.utcnow()

    updated_user = await update_user_in_db(current_user)

    return UpdateUserResponse(
        user=UserResponse.from_orm(updated_user),
        message="Avatar updated successfully"
    )

@router.put("/update-cover", response_model=UpdateUserResponse)
async def update_user_cover_image(
    cover_data: UpdateUserCoverImageRequest,
```

```
    current_user: UserInDB = Depends(get_current_user)
):
    """Update user cover image"""
    current_user.cover_image_url = cover_data.cover_image_url
    current_user.updated_at = datetime.utcnow()

    updated_user = await update_user_in_db(current_user)

    return UpdateUserResponse(
        user=UserResponse.from_orm(updated_user),
        message="Cover image updated successfully"
    )
```

🎯 Best Practices

1. Model Validation

```
from pydantic import validator, root_validator

class UserModel(BaseModel):
    email: EmailStr
    age: int

    @validator('age')
    def validate_age(cls, v):
        if v < 13:
            raise ValueError('User must be at least 13 years old')
        return v

    @root_validator
    def validate_model(cls, values):
        # Cross-field validation
        return values
```

2. Field Configuration

```
from pydantic import Field

class UserModel(BaseModel):
    username: str = Field(
        ...,
        min_length=3,
        max_length=20,
        regex=r'^[a-zA-Z0-9_]+$',
        description="Unique username",
```

```
        example="johndoe"
    )
```

3. Model Configuration

```
class UserModel(BaseModel):
    name: str

    class Config:
        # Allow field aliases
        allow_population_by_field_name = True
        # Validate on assignment
        validate_assignment = True
        # Use enum values
        use_enum_values = True
        # JSON schema extra
        schema_extra = {
            "example": {
                "name": "John Doe"
            }
        }
```

4. Error Handling

```
from fastapi import HTTPException, status
from pydantic import ValidationError

try:
    user = UserModel(**data)
except ValidationError as e:
    raise HTTPException(
        status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
        detail=e.errors()
    )
```

🚀 Key Takeaways

1. **Always inherit from BaseModel** for FastAPI integration
2. **Use 3-layer model architecture** for clean separation
3. **Implement proper validation** at the Pydantic level
4. **Never expose sensitive data** in response models
5. **Use proper field configuration** for better documentation
6. **Implement consistent error handling** across models
7. **Structure your project** for scalability and maintainability

This comprehensive guide provides a solid foundation for building robust FastAPI applications with proper model management and user authentication systems.