FastAPI Hero Tutorial: From Zero to Expert

A Complete Guide with Practice Exercises

Table of Contents

Introduction & Setup

What is FastAPI?

FastAPI is a modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints. It's designed to be easy to use, fast to code, ready for production, and based on standards like OpenAPI and JSON Schema.

Key Features

- **Fast**: Very high performance, on par with NodeJS and Go
- **Fast to code**: Increase development speed by 200-300%
- **Fewer bugs**: Reduce human errors by 40%
- **Intuitive**: Great editor support with auto-completion
- **Easy**: Designed to be easy to use and learn
- **Short**: Minimize code duplication
- **Robust**: Production-ready code with automatic interactive documentation
- **Standards-based**: Based on OpenAPI and JSON Schema

Installation

bash

```
# Create a virtual environment
python -m venv fastapi-env

# Activate it (Windows)
fastapi-env\Scripts\activate

# Activate it (Mac/Linux)
source fastapi-env/bin/activate

# Install FastAPI and Uvicorn
pip install fastapi uvicorn[standard]

# Additional packages we'll use
pip install python-multipart sqlalchemy python-jose[cryptography] passlib[bcrypt] python-dotenv
```

Chapter 1: Your First FastAPI Application

Basic Application Structure

python

```python
# main.py
from fastapi import FastAPI

# Create FastAPI instance
app = FastAPI(
    title="My First API",
    description="Learning FastAPI like a hero",
    version="1.0.0"
)

# Root endpoint
@app.get("/")
def read_root():
    return {"message": "Hello, FastAPI Hero!"}

# Path parameter example
@app.get("/items/{item_id}")
def read_item(item_id: int):
    return {"item_id": item_id, "message": f"You requested item {item_id}"}

# Query parameter example
@app.get("/users")
def read_users(skip: int = 0, limit: int = 10):
    return {"skip": skip, "limit": limit}
```

Running the Application

```bash
# Run with auto-reload for development
uvicorn main:app --reload

# Your API is now available at http://localhost:8000
# Interactive docs at http://localhost:8000/docs
# Alternative docs at http://localhost:8000/redoc
```

Key Concepts Explained

Decorators: `@app.get("/")` tells FastAPI that the function below handles GET requests to the "/" path.

Type Hints: FastAPI uses Python type hints for:

- Editor support (auto-completion, error checks)
- Data validation
- Automatic API documentation
- Type conversion

Automatic Documentation: FastAPI generates interactive API documentation automatically using Swagger UI (`/docs`) and ReDoc (`/redoc`).

📝 Practice Exercise 1

Create an API with the following endpoints:

1. GET `/` - Returns a welcome message

2. GET `/about` - Returns information about your API
3. GET `/calculate/{num1}/{num2}` - Returns the sum of two numbers
4. GET `/greet` - Accepts a query parameter `name` and returns a greeting

Solution:

```python
```

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def home():
    return {"message": "Welcome to my practice API!"}

@app.get("/about")
def about():
    return {
        "name": "Practice API",
        "version": "1.0",
        "description": "Learning FastAPI step by step"
    }

@app.get("/calculate/{num1}/{num2}")
def calculate(num1: int, num2: int):
    return {
        "num1": num1,
        "num2": num2,
        "sum": num1 + num2,
        "product": num1 * num2
    }

@app.get("/greet")
def greet(name: str = "World"):
    return {"greeting": f"Hello, {name}!"}
```

Chapter 2: Request Handling & Data Validation

Pydantic Models

Pydantic is FastAPI's data validation library. It ensures your data is correct and converts it to the right type.

```python
```

```python
from fastapi import FastAPI
from pydantic import BaseModel, Field, EmailStr
from typing import Optional, List
from datetime import datetime
from enum import Enum

app = FastAPI()

# Enum for categories
class CategoryEnum(str, Enum):
    electronics = "electronics"
    clothing = "clothing"
    food = "food"
    books = "books"

# Pydantic model for request body
class Item(BaseModel):
    name: str = Field(..., min_length=1, max_length=100, description="Item name")
    price: float = Field(..., gt=0, description="Price must be greater than 0")
    description: Optional[str] = Field(None, max_length=500)
    tax: Optional[float] = None
    category: CategoryEnum
    tags: List[str] = []
    in_stock: bool = True

    # Config with example
    class Config:
        json_schema_extra = {
            "example": {
                "name": "Laptop",
                "price": 999.99,
                "description": "High-performance laptop",
```

```
            "tax": 99.99,
            "category": "electronics",
            "tags": ["computers", "electronics"],
            "in_stock": True
        }
    }

# POST endpoint with request body
@app.post("/items/")
def create_item(item: Item):
    total_price = item.price + (item.tax or 0)
    return {
        "message": "Item created successfully",
        "item": item,
        "total_price": total_price
    }

# PUT endpoint for updates
@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item):
    return {
        "item_id": item_id,
        "updated_item": item,
        "timestamp": datetime.now()
    }

# Mixed parameters: path, query, and body
@app.post("/items/{item_id}/reviews")
def create_review(
    item_id: int,
    review: str,
    rating: int = Field(..., ge=1, le=5),
```

```
    reviewer_name: Optional[str] = None
):
    return {
        "item_id": item_id,
        "review": review,
        "rating": rating,
        "reviewer": reviewer_name or "Anonymous"
    }
```

Validation Features

Field Validators:

- `min_length`, `max_length`: String length constraints
- `gt`, `ge`, `lt`, `le`: Numeric comparisons (greater than, greater equal, etc.)
- `regex`: Pattern matching for strings
- `EmailStr`: Email validation (requires `pip install email-validator`)

Request Body with Nested Models

```python
```

```python
from typing import Dict

class Address(BaseModel):
    street: str
    city: str
    country: str
    zip_code: str = Field(..., regex="^\\d{5}$")

class User(BaseModel):
    username: str = Field(..., min_length=3, max_length=20)
    email: EmailStr
    full_name: Optional[str] = None
    age: int = Field(..., ge=0, le=120)
    address: Address
    metadata: Dict[str, str] = {}

@app.post("/users/")
def create_user(user: User):
    return {"message": f"User {user.username} created", "user": user}
```

📝 Practice Exercise 2

Create a book management API with:

1. A `Book` model with: title, author, isbn, pages, published_year, genres (list), available (bool)
2. POST `/books/` - Create a new book
3. GET `/books/search` - Search books by author (query parameter)
4. PUT `/books/{isbn}` - Update book availability

Solution:

python

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field
from typing import List, Optional

app = FastAPI()

class Book(BaseModel):
    title: str = Field(..., min_length=1, max_length=200)
    author: str = Field(..., min_length=1, max_length=100)
    isbn: str = Field(..., regex="^\\d{13}$")
    pages: int = Field(..., gt=0)
    published_year: int = Field(..., ge=1900, le=2024)
    genres: List[str] = []
    available: bool = True

# In-memory storage
books_db = {}

@app.post("/books/")
def create_book(book: Book):
    if book.isbn in books_db:
        raise HTTPException(status_code=400, detail="Book already exists")
    books_db[book.isbn] = book
    return {"message": "Book created", "book": book}

@app.get("/books/search")
def search_books(author: str):
    found_books = [
        book for book in books_db.values()
        if book.author.lower() == author.lower()
    ]
    return {"author": author, "books": found_books}
```

```python
@app.put("/books/{isbn}")
def update_availability(isbn: str, available: bool):
    if isbn not in books_db:
        raise HTTPException(status_code=404, detail="Book not found")
    books_db[isbn].available = available
    return {"message": "Updated", "book": books_db[isbn]}
```

Chapter 3: Response Models & Status Codes

Response Models

Control exactly what data is returned to the client:

```python
```

```python
from fastapi import FastAPI, status
from pydantic import BaseModel, EmailStr
from typing import Optional, List

app = FastAPI()

# Input model (what user sends)
class UserIn(BaseModel):
    username: str
    email: EmailStr
    password: str
    full_name: Optional[str] = None

# Output model (what API returns - no password!)
class UserOut(BaseModel):
    id: int
    username: str
    email: EmailStr
    full_name: Optional[str] = None
    is_active: bool = True

# Database model (internal use)
class UserInDB(UserOut):
    hashed_password: str

# Simulate database
fake_users_db = {}
user_counter = 1

def fake_hash_password(password: str):
    return f"hashed_{password}"
```

```python
@app.post(
    "/users/",
    response_model=UserOut,
    status_code=status.HTTP_201_CREATED,
    tags=["users"],
    summary="Create a new user",
    response_description="The created user"
)
def create_user(user: UserIn):
    global user_counter

    # Create user in database
    db_user = UserInDB(
        id=user_counter,
        username=user.username,
        email=user.email,
        full_name=user.full_name,
        hashed_password=fake_hash_password(user.password)
    )

    fake_users_db[user_counter] = db_user
    user_counter += 1

    # Return only safe fields (response_model handles this)
    return db_user

# Response model with List
@app.get("/users/", response_model=List[UserOut])
def get_all_users():
    return list(fake_users_db.values())

# Exclude unset values
```

```python
@app.get(
    "/users/{user_id}",
    response_model=UserOut,
    response_model_exclude_unset=True
)
def get_user(user_id: int):
    if user_id not in fake_users_db:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="User not found"
        )
    return fake_users_db[user_id]
```

Status Codes

python

```python
from fastapi import FastAPI, status, HTTPException

app = FastAPI()

# Common status codes:
# 200: OK (default for GET)
# 201: Created (for POST when creating resources)
# 204: No Content (successful but no response body)
# 400: Bad Request
# 401: Unauthorized
# 403: Forbidden
# 404: Not Found
# 422: Validation Error (FastAPI default for invalid data)
# 500: Internal Server Error

@app.delete("/items/{item_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_item(item_id: int):
    # Delete logic here
    return  # No content returned for 204

@app.post("/login", status_code=status.HTTP_200_OK)
def login(username: str, password: str):
    if username != "admin" or password != "secret":
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid credentials",
            headers={"WWW-Authenticate": "Bearer"},
        )
    return {"access_token": "fake-token"}
```

📝 Practice Exercise 3

Create a product API with proper response models:

1. `ProductCreate` model (input): name, price, description
2. `ProductPublic` model (output): id, name, price (no description)
3. `ProductInternal` model: all fields plus created_at
4. POST `/products/` - Returns `ProductPublic`, status 201
5. GET `/products/` - Returns list of `ProductPublic`
6. Handle 404 errors properly

Solution:

```python
```

```python
from fastapi import FastAPI, HTTPException, status
from pydantic import BaseModel
from typing import List
from datetime import datetime

app = FastAPI()

class ProductCreate(BaseModel):
    name: str
    price: float
    description: str

class ProductPublic(BaseModel):
    id: int
    name: str
    price: float

class ProductInternal(ProductPublic):
    description: str
    created_at: datetime

products_db = {}
product_counter = 1

@app.post(
    "/products/",
    response_model=ProductPublic,
    status_code=status.HTTP_201_CREATED
)
def create_product(product: ProductCreate):
    global product_counter
```

```
    internal_product = ProductInternal(
        id=product_counter,
        name=product.name,
        price=product.price,
        description=product.description,
        created_at=datetime.now()
    )

    products_db[product_counter] = internal_product
    product_counter += 1

    return internal_product

@app.get("/products/", response_model=List[ProductPublic])
def get_products():
    return list(products_db.values())

@app.get("/products/{product_id}", response_model=ProductPublic)
def get_product(product_id: int):
    if product_id not in products_db:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Product with id {product_id} not found"
        )
    return products_db[product_id]
```

Chapter 4: Dependency Injection with Depends

Understanding Depends

`Depends` is FastAPI's dependency injection system. It allows you to:

- Share logic between endpoints
- Manage database connections
- Handle authentication
- Validate data
- Reduce code duplication

Basic Dependencies

```python
```

```python
from fastapi import FastAPI, Depends, HTTPException, Query
from typing import Optional

app = FastAPI()

# Simple dependency function
def common_parameters(
    q: Optional[str] = None,
    skip: int = 0,
    limit: int = Query(default=10, le=100)
):
    return {"q": q, "skip": skip, "limit": limit}

# Using the dependency
@app.get("/items/")
def read_items(commons: dict = Depends(common_parameters)):
    return {"message": "Reading items", "params": commons}

@app.get("/users/")
def read_users(commons: dict = Depends(common_parameters)):
    return {"message": "Reading users", "params": commons}
```

Class-based Dependencies

python

```python
from fastapi import Depends

class CommonQueryParams:
    def __init__(
        self,
        q: Optional[str] = None,
        skip: int = 0,
        limit: int = 10
    ):
        self.q = q
        self.skip = skip
        self.limit = limit

@app.get("/products/")
def get_products(params: CommonQueryParams = Depends()):
    return {
        "query": params.q,
        "pagination": {"skip": params.skip, "limit": params.limit}
    }
```

Nested Dependencies

python

```python
from fastapi import Depends, Header, HTTPException

# First level dependency
def verify_token(x_token: str = Header()):
    if x_token != "fake-super-secret-token":
        raise HTTPException(status_code=400, detail="Invalid X-Token header")
    return x_token

# Second level dependency that uses the first
def get_current_user(token: str = Depends(verify_token)):
    # In real app, decode token and get user from database
    return {"username": "john_doe", "email": "john@example.com"}

# Using nested dependencies
@app.get("/protected/")
def protected_route(current_user: dict = Depends(get_current_user)):
    return {"message": f"Hello {current_user['username']}!"}
```

Database Session Dependency

python

```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, Session

# Database setup
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

# Dependency to get DB session
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# Using DB dependency
@app.post("/items/")
def create_item(item: dict, db: Session = Depends(get_db)):
    # Use db session here
    # db.add(item)
    # db.commit()
    return {"message": "Item would be saved to database"}
```

Path Operation Dependencies

python

```python
# Dependencies that don't return values (e.g., for validation)
def verify_admin(x_admin: str = Header()):
    if x_admin != "true":
        raise HTTPException(status_code=403, detail="Admin access required")

# Apply to specific endpoint
@app.delete(
    "/admin/users/{user_id}",
    dependencies=[Depends(verify_admin)]
)
def delete_user(user_id: int):
    return {"message": f"User {user_id} deleted"}

# Apply to all endpoints in the app
app = FastAPI(dependencies=[Depends(verify_token)])
```

Advanced Dependency Patterns

python

```python
from functools import lru_cache
from pydantic import BaseSettings

# Settings with caching
class Settings(BaseSettings):
    app_name: str = "FastAPI Hero App"
    debug: bool = False
    database_url: str = "sqlite:///./test.db"
    secret_key: str = "your-secret-key"

@lru_cache()
def get_settings():
    return Settings()

# Rate limiting dependency
from datetime import datetime, timedelta
from collections import defaultdict

request_counts = defaultdict(list)

def rate_limit(max_requests: int = 10, window: int = 60):
    def rate_limit_dependency(request: Request):
        client_ip = request.client.host
        now = datetime.now()

        # Clean old requests
        request_counts[client_ip] = [
            req_time for req_time in request_counts[client_ip]
            if req_time > now - timedelta(seconds=window)
        ]

        if len(request_counts[client_ip]) >= max_requests:
```

```
            raise HTTPException(
                status_code=429,
                detail="Too many requests"
            )

        request_counts[client_ip].append(now)

    return rate_limit_dependency

@app.get("/limited/")
def limited_endpoint(
    _: None = Depends(rate_limit(max_requests=5, window=60))
):
    return {"message": "This endpoint is rate limited"}
```

📝 Practice Exercise 4

Create an API with proper dependency injection:

1. Create a `verify_api_key` dependency that checks for an API key in headers
2. Create a `PaginationParams` class dependency with page and size parameters
3. Create a `get_current_user` dependency that depends on `verify_api_key`
4. Apply these to endpoints that list and create posts

Solution:

```
python
```

```python
from fastapi import FastAPI, Depends, Header, HTTPException, status
from pydantic import BaseModel
from typing import Optional, List

app = FastAPI()

# API Key dependency
def verify_api_key(x_api_key: str = Header()):
    valid_keys = ["key123", "key456"]
    if x_api_key not in valid_keys:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid API Key"
        )
    return x_api_key

# Pagination dependency
class PaginationParams:
    def __init__(self, page: int = 1, size: int = 10):
        self.page = max(1, page)
        self.size = min(100, max(1, size))
        self.skip = (self.page - 1) * self.size

# User dependency (depends on API key)
def get_current_user(api_key: str = Depends(verify_api_key)):
    # Simulate user lookup based on API key
    users = {
        "key123": {"id": 1, "username": "alice"},
        "key456": {"id": 2, "username": "bob"}
    }
    return users.get(api_key)
```

```
# Post model
class Post(BaseModel):
    title: str
    content: str
    author_id: Optional[int] = None

posts_db = []

@app.get("/posts/", response_model=List[Post])
def list_posts(
    pagination: PaginationParams = Depends(),
    current_user: dict = Depends(get_current_user)
):
    start = pagination.skip
    end = start + pagination.size
    return posts_db[start:end]

@app.post("/posts/", response_model=Post, status_code=status.HTTP_201_CREATED)
def create_post(
    post: Post,
    current_user: dict = Depends(get_current_user)
):
    post.author_id = current_user["id"]
    posts_db.append(post)
    return post

@app.get("/me/")
def get_me(current_user: dict = Depends(get_current_user)):
    return current_user
```

Chapter 5: Database Integration

SQLAlchemy Setup

python

```python
# database.py
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./fastapi_hero.db"
# For PostgreSQL: "postgresql://user:password@localhost/dbname"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL,
    connect_args={"check_same_thread": False}  # Only for SQLite
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Database Models

python

```python
# models.py
from sqlalchemy import Column, Integer, String, Float, Boolean, DateTime, ForeignKey, Table
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from database import Base

# Many-to-many relationship table
post_tags = Table(
    'post_tags',
    Base.metadata,
    Column('post_id', Integer, ForeignKey('posts.id')),
    Column('tag_id', Integer, ForeignKey('tags.id'))
)

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True, nullable=False)
    username = Column(String, unique=True, index=True, nullable=False)
    hashed_password = Column(String, nullable=False)
    is_active = Column(Boolean, default=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())

    # Relationship
    posts = relationship("Post", back_populates="author")

class Post(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, nullable=False)
```

```python
    content = Column(String, nullable=False)
    published = Column(Boolean, default=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())

    # Foreign key
    author_id = Column(Integer, ForeignKey("users.id"))

    # Relationships
    author = relationship("User", back_populates="posts")
    tags = relationship("Tag", secondary=post_tags, back_populates="posts")

class Tag(Base):
    __tablename__ = "tags"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, unique=True, index=True)

    # Relationship
    posts = relationship("Post", secondary=post_tags, back_populates="tags")
```

Pydantic Schemas

```python
```

```python
# schemas.py
from pydantic import BaseModel, EmailStr
from typing import Optional, List
from datetime import datetime

# User schemas
class UserBase(BaseModel):
    email: EmailStr
    username: str

class UserCreate(UserBase):
    password: str

class UserUpdate(BaseModel):
    email: Optional[EmailStr] = None
    username: Optional[str] = None
    password: Optional[str] = None

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

    class Config:
        from_attributes = True

# Post schemas
class PostBase(BaseModel):
    title: str
    content: str
    published: bool = True
```

```python
class PostCreate(PostBase):
    pass

class Post(PostBase):
    id: int
    created_at: datetime
    author_id: int
    author: User

    class Config:
        from_attributes = True

# Tag schemas
class TagBase(BaseModel):
    name: str

class TagCreate(TagBase):
    pass

class Tag(TagBase):
    id: int

    class Config:
        from_attributes = True
```

CRUD Operations

python

```python
# crud.py
from sqlalchemy.orm import Session
from sqlalchemy import or_
import models, schemas
from passlib.context import CryptContext

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

# User CRUD
def get_user(db: Session, user_id: int):
    return db.query(models.User).filter(models.User.id == user_id).first()

def get_user_by_email(db: Session, email: str):
    return db.query(models.User).filter(models.User.email == email).first()

def get_users(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.User).offset(skip).limit(limit).all()

def create_user(db: Session, user: schemas.UserCreate):
    hashed_password = pwd_context.hash(user.password)
    db_user = models.User(
        email=user.email,
        username=user.username,
        hashed_password=hashed_password
    )
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user

def update_user(db: Session, user_id: int, user_update: schemas.UserUpdate):
    db_user = get_user(db, user_id)
```

```python
    if not db_user:
        return None

    update_data = user_update.dict(exclude_unset=True)
    if "password" in update_data:
        update_data["hashed_password"] = pwd_context.hash(update_data.pop("password"))

    for field, value in update_data.items():
        setattr(db_user, field, value)

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

def delete_user(db: Session, user_id: int):
    db_user = get_user(db, user_id)
    if db_user:
        db.delete(db_user)
        db.commit()
        return True
    return False

# Post CRUD
def get_posts(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.Post).offset(skip).limit(limit).all()

def create_post(db: Session, post: schemas.PostCreate, user_id: int):
    db_post = models.Post(**post.dict(), author_id=user_id)
    db.add(db_post)
    db.commit()
    db.refresh(db_post)
    return db_post
```

```
def search_posts(db: Session, query: str):
    return db.query(models.Post).filter(
        or_(
            models.Post.title.contains(query),
            models.Post.content.contains(query)
        )
    ).all()
```

API Endpoints with Database

python

```python
# main.py
from fastapi import FastAPI, Depends, HTTPException, status
from sqlalchemy.orm import Session
from typing import List

import models, schemas, crud
from database import engine, get_db

# Create tables
models.Base.metadata.create_all(bind=engine)

app = FastAPI(title="FastAPI with Database")

# User endpoints
@app.post("/users/", response_model=schemas.User)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):
    db_user = crud.get_user_by_email(db, email=user.email)
    if db_user:
        raise HTTPException(
            status_code=400,
            detail="Email already registered"
        )
    return crud.create_user(db=db, user=user)

@app.get("/users/", response_model=List[schemas.User])
def read_users(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    users = crud.get_users(db, skip=skip, limit=limit)
    return users

@app.get("/users/{user_id}", response_model=schemas.User)
def read_user(user_id: int, db: Session = Depends(get_db)):
    db_user = crud.get_user(db, user_id=user_id)
```

```
    if db_user is None:
        raise HTTPException(status_code=404, detail="User not found")
    return db_user

@app.put("/users/{user_id}", response_model=schemas.User)
def update_user(
    user_id: int,
    user_update: schemas.UserUpdate,
    db: Session = Depends(get_db)
):
    db_user = crud.update_user(db, user_id, user_update)
    if db_user is None:
        raise HTTPException(status_code=404, detail="User not found")
    return db_user

@app.delete("/users/{user_id}")
def delete_user(user_id: int, db: Session = Depends(get_db)):
    success = crud.delete_user(db, user_id)
    if not success:
        raise HTTPException(status_code=404, detail="User not found")
    return {"message": "User deleted successfully"}

# Post endpoints
@app.post("/users/{user_id}/posts/", response_model=schemas.Post)
def create_post(
    user_id: int,
    post: schemas.PostCreate,
    db: Session = Depends(get_db)
):
    return crud.create_post(db=db, post=post, user_id=user_id)

@app.get("/posts/", response_model=List[schemas.Post])
```

```python
def read_posts(
    skip: int = 0,
    limit: int = 100,
    db: Session = Depends(get_db)
):
    posts = crud.get_posts(db, skip=skip, limit=limit)
    return posts

@app.get("/posts/search", response_model=List[schemas.Post])
def search_posts(q: str, db: Session = Depends(get_db)):
    return crud.search_posts(db, query=q)
```

📝 Practice Exercise 5

Create a Todo application with database:

1. Create a `Todo` model with: id, title, description, completed, created_at
2. Create CRUD operations for todos
3. Add endpoints: create, list, update, delete, mark as complete
4. Add search functionality

Solution:

```python
```

```python
# models.py addition
class Todo(Base):
    __tablename__ = "todos"

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, nullable=False)
    description = Column(String)
    completed = Column(Boolean, default=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    user_id = Column(Integer, ForeignKey("users.id"))

    owner = relationship("User")

# schemas.py addition
class TodoBase(BaseModel):
    title: str
    description: Optional[str] = None
    completed: bool = False

class TodoCreate(TodoBase):
    pass

class TodoUpdate(BaseModel):
    title: Optional[str] = None
    description: Optional[str] = None
    completed: Optional[bool] = None

class Todo(TodoBase):
    id: int
    created_at: datetime
    user_id: int
```

```
    class Config:
        from_attributes = True

# crud.py addition
def create_todo(db: Session, todo: schemas.TodoCreate, user_id: int):
    db_todo = models.Todo(**todo.dict(), user_id=user_id)
    db.add(db_todo)
    db.commit()
    db.refresh(db_todo)
    return db_todo

def get_todos(db: Session, user_id: int, skip: int = 0, limit: int = 100):
    return db.query(models.Todo).filter(
        models.Todo.user_id == user_id
    ).offset(skip).limit(limit).all()

def update_todo(db: Session, todo_id: int, todo_update: schemas.TodoUpdate):
    db_todo = db.query(models.Todo).filter(models.Todo.id == todo_id).first()
    if not db_todo:
        return None

    for field, value in todo_update.dict(exclude_unset=True).items():
        setattr(db_todo, field, value)

    db.commit()
    db.refresh(db_todo)
    return db_todo

def delete_todo(db: Session, todo_id: int):
    db_todo = db.query(models.Todo).filter(models.Todo.id == todo_id).first()
    if db_todo:
        db.delete(db_todo)
```

```
        db.commit()
        return True
    return False

# main.py addition
@app.post("/todos/", response_model=schemas.Todo)
def create_todo(
    todo: schemas.TodoCreate,
    db: Session = Depends(get_db),
    current_user: dict = {"id": 1}  # Simplified auth
):
    return crud.create_todo(db=db, todo=todo, user_id=current_user["id"])

@app.get("/todos/", response_model=List[schemas.Todo])
def read_todos(
    skip: int = 0,
    limit: int = 100,
    db: Session = Depends(get_db),
    current_user: dict = {"id": 1}
):
    return crud.get_todos(db, user_id=current_user["id"], skip=skip, limit=limit)

@app.put("/todos/{todo_id}", response_model=schemas.Todo)
def update_todo(
    todo_id: int,
    todo_update: schemas.TodoUpdate,
    db: Session = Depends(get_db)
):
    db_todo = crud.update_todo(db, todo_id, todo_update)
    if not db_todo:
        raise HTTPException(status_code=404, detail="Todo not found")
    return db_todo
```

```python
@app.delete("/todos/{todo_id}")
def delete_todo(todo_id: int, db: Session = Depends(get_db)):
    success = crud.delete_todo(db, todo_id)
    if not success:
        raise HTTPException(status_code=404, detail="Todo not found")
    return {"message": "Todo deleted"}

@app.patch("/todos/{todo_id}/complete")
def mark_complete(todo_id: int, db: Session = Depends(get_db)):
    todo_update = schemas.TodoUpdate(completed=True)
    db_todo = crud.update_todo(db, todo_id, todo_update)
    if not db_todo:
        raise HTTPException(status_code=404, detail="Todo not found")
    return {"message": "Todo marked as complete"}
```

Chapter 6: Authentication & Security

JWT Authentication

python

```python
# auth.py
from datetime import datetime, timedelta
from typing import Optional
from jose import JWTError, jwt
from passlib.context import CryptContext
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from sqlalchemy.orm import Session
from pydantic import BaseModel

# Security configuration
SECRET_KEY = "your-secret-key-here-change-in-production"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

# Token schemas
class Token(BaseModel):
    access_token: str
    token_type: str

class TokenData(BaseModel):
    username: Optional[str] = None

# Password utilities
def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password):
    return pwd_context.hash(password)
```

```python
# JWT utilities
def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

def verify_token(token: str, credentials_exception):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
        token_data = TokenData(username=username)
        return token_data
    except JWTError:
        raise credentials_exception

# Authentication functions
def authenticate_user(db: Session, username: str, password: str):
    user = crud.get_user_by_username(db, username)
    if not user:
        return False
    if not verify_password(password, user.hashed_password):
        return False
    return user
```

```python
async def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: Session = Depends(get_db)
):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    token_data = verify_token(token, credentials_exception)
    user = crud.get_user_by_username(db, username=token_data.username)
    if user is None:
        raise credentials_exception
    return user

async def get_current_active_user(
    current_user: models.User = Depends(get_current_user)
):
    if not current_user.is_active:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user
```

Secured Endpoints

python

```python
# main.py additions
from fastapi.security import OAuth2PasswordRequestForm
from auth import (
    authenticate_user, create_access_token, get_current_active_user,
    Token, ACCESS_TOKEN_EXPIRE_MINUTES
)

# Login endpoint
@app.post("/token", response_model=Token)
async def login(
    form_data: OAuth2PasswordRequestForm = Depends(),
    db: Session = Depends(get_db)
):
    user = authenticate_user(db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username}, expires_delta=access_token_expires
    )
    return {"access_token": access_token, "token_type": "bearer"}

# Protected endpoints
@app.get("/users/me", response_model=schemas.User)
async def read_users_me(current_user: models.User = Depends(get_current_active_user)):
    return current_user

@app.post("/posts/", response_model=schemas.Post)
```

```python
async def create_post(
    post: schemas.PostCreate,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(get_current_active_user)
):
    return crud.create_post(db=db, post=post, user_id=current_user.id)

# Admin only endpoint
def get_admin_user(current_user: models.User = Depends(get_current_active_user)):
    if not current_user.is_admin:  # Assuming you add is_admin field
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Not enough permissions"
        )
    return current_user

@app.delete("/admin/users/{user_id}")
async def admin_delete_user(
    user_id: int,
    db: Session = Depends(get_db),
    admin_user: models.User = Depends(get_admin_user)
):
    success = crud.delete_user(db, user_id)
    if not success:
        raise HTTPException(status_code=404, detail="User not found")
    return {"message": "User deleted"}
```

Role-Based Access Control (RBAC)

python

```python
# models.py addition
from enum import Enum as PyEnum

class UserRole(PyEnum):
    USER = "user"
    MODERATOR = "moderator"
    ADMIN = "admin"

# Update User model
class User(Base):
    # ... existing fields ...
    role = Column(Enum(UserRole), default=UserRole.USER)

# auth.py addition
def require_role(allowed_roles: List[UserRole]):
    def role_checker(current_user: models.User = Depends(get_current_active_user)):
        if current_user.role not in allowed_roles:
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="Operation not permitted"
            )
        return current_user
    return role_checker

# Usage in endpoints
@app.delete("/moderator/posts/{post_id}")
async def moderator_delete_post(
    post_id: int,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(
        require_role([UserRole.MODERATOR, UserRole.ADMIN])
    )
```

```
):
    # Delete post logic
    return {"message": "Post deleted"}
```

API Key Authentication

python

```python
from fastapi import Security, HTTPException
from fastapi.security import APIKeyHeader, APIKeyQuery, APIKeyCookie

# Multiple API key options
api_key_header = APIKeyHeader(name="X-API-Key", auto_error=False)
api_key_query = APIKeyQuery(name="api_key", auto_error=False)
api_key_cookie = APIKeyCookie(name="api_key", auto_error=False)

async def get_api_key(
    api_key_header: str = Security(api_key_header),
    api_key_query: str = Security(api_key_query),
    api_key_cookie: str = Security(api_key_cookie),
):
    if api_key_header == "secret-api-key":
        return api_key_header
    elif api_key_query == "secret-api-key":
        return api_key_query
    elif api_key_cookie == "secret-api-key":
        return api_key_cookie
    else:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Could not validate API key"
        )

@app.get("/api-key-protected")
async def protected_route(api_key: str = Depends(get_api_key)):
    return {"message": "Access granted", "api_key": api_key}
```

📝 Practice Exercise 6

Build a secure blog API with:

1. User registration with password hashing
2. JWT login system
3. Protected endpoints for creating/editing posts
4. Role-based access (USER, AUTHOR, ADMIN)
5. Only authors can edit their own posts
6. Admins can edit/delete any post

Solution:

```python
```

```python
# Complete authentication system
from enum import Enum as PyEnum

class UserRole(PyEnum):
    USER = "user"
    AUTHOR = "author"
    ADMIN = "admin"

# models.py
class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    username = Column(String, unique=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)
    role = Column(Enum(UserRole), default=UserRole.USER)

    posts = relationship("BlogPost", back_populates="author")

class BlogPost(Base):
    __tablename__ = "blog_posts"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    content = Column(String)
    published = Column(Boolean, default=False)
    author_id = Column(Integer, ForeignKey("users.id"))

    author = relationship("User", back_populates="posts")
```

```python
# schemas.py
class UserRegister(BaseModel):
    username: str
    email: EmailStr
    password: str
    role: UserRole = UserRole.USER

class BlogPostCreate(BaseModel):
    title: str
    content: str
    published: bool = False

class BlogPostUpdate(BaseModel):
    title: Optional[str] = None
    content: Optional[str] = None
    published: Optional[bool] = None

# main.py
@app.post("/register", response_model=schemas.User)
def register(
    user: schemas.UserRegister,
    db: Session = Depends(get_db)
):
    # Check if user exists
    existing = db.query(models.User).filter(
        (models.User.username == user.username) |
        (models.User.email == user.email)
    ).first()

    if existing:
        raise HTTPException(status_code=400, detail="User already exists")
```

```
    # Create new user
    hashed_password = get_password_hash(user.password)
    db_user = models.User(
        username=user.username,
        email=user.email,
        hashed_password=hashed_password,
        role=user.role
    )
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user

@app.post("/posts/", response_model=schemas.BlogPost)
def create_post(
    post: schemas.BlogPostCreate,
    db: Session = Depends(get_db),
    current_user: models.User = Depends(
        require_role([UserRole.AUTHOR, UserRole.ADMIN])
    )
):
    db_post = models.BlogPost(**post.dict(), author_id=current_user.id)
    db.add(db_post)
    db.commit()
    db.refresh(db_post)
    return db_post

@app.put("/posts/{post_id}")
def update_post(
    post_id: int,
    post_update: schemas.BlogPostUpdate,
    db: Session = Depends(get_db),
```

```
    current_user: models.User = Depends(get_current_active_user)
):
    db_post = db.query(models.BlogPost).filter(
        models.BlogPost.id == post_id
    ).first()

    if not db_post:
        raise HTTP
```