Core Concepts & Architecture

Application Structure

FastAPI applications are built on ASGI (Asynchronous Server Gateway Interface), using Starlette for web parts and Pydantic for data validation.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

Key architectural principles:

- Path operations are Python functions decorated with HTTP method decorators
- Async/await support is native but optional (sync functions work too)
- Automatic OpenAPI schema generation
- Built-in request validation via type hints

Path Operations & Routing

HTTP Methods

```
@app.get("/items/{item_id}")          # Read
@app.post("/items")                    # Create
@app.put("/items/{item_id}")           # Update (full)
@app.patch("/items/{item_id}")         # Update (partial)
@app.delete("/items/{item_id}")        # Delete
```

Path Parameters

```
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}

# Path parameter with enum validation
from enum import Enum

class ModelName(str, Enum):
```

```
    alexnet = "alexnet"
    resnet = "resnet"

@app.get("/models/{model_name}")
async def get_model(model_name: ModelName):
    return {"model_name": model_name}
```

Query Parameters

```
@app.get("/items/")
async def read_items(skip: int = 0, limit: int = 10):
    return {"skip": skip, "limit": limit}

# Optional query parameters
from typing import Optional

@app.get("/items/{item_id}")
async def read_item(item_id: str, q: Optional[str] = None):
    if q:
        return {"item_id": item_id, "q": q}
    return {"item_id": item_id}
```

Request Body & Pydantic Models

Basic Model Definition

```
from pydantic import BaseModel, Field

class Item(BaseModel):
    name: str
    description: Optional[str] = Field(None, max_length=300)
    price: float = Field(gt=0, description="Price must be greater than
zero")
    tax: Optional[float] = None

@app.post("/items/")
async def create_item(item: Item):
    return item
```

Nested Models

```
class Image(BaseModel):
    url: str
    name: str
```

```python
class Item(BaseModel):
    name: str
    images: Optional[List[Image]] = None

@app.post("/items/")
async def create_item(item: Item):
    return item
```

Model Configuration

```python
class Item(BaseModel):
    name: str
    price: float

    class Config:
        schema_extra = {
            "example": {
                "name": "Foo",
                "price": 35.4
            }
        }
```

Dependency Injection

Basic Dependencies

```python
from fastapi import Depends

def common_parameters(q: Optional[str] = None, skip: int = 0, limit: int =
100):
    return {"q": q, "skip": skip, "limit": limit}

@app.get("/items/")
async def read_items(commons: dict = Depends(common_parameters)):
    return commons
```

Class-based Dependencies

```python
class CommonQueryParams:
    def __init__(self, q: Optional[str] = None, skip: int = 0, limit: int =
100):
        self.q = q
        self.skip = skip
```

```
        self.limit = limit

@app.get("/items/")
async def read_items(commons: CommonQueryParams = Depends()):
    return commons
```

Sub-dependencies

```
def query_extractor(q: Optional[str] = None):
    return q

def query_or_cookie_extractor(
    q: str = Depends(query_extractor),
    last_query: Optional[str] = Cookie(None)
):
    if not q:
        return last_query
    return q

@app.get("/items/")
async def read_query(query_or_default: str =
Depends(query_or_cookie_extractor)):
    return {"q_or_cookie": query_or_default}
```

Database Integration (SQLAlchemy)

Database Setup

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./sql_app.db"
# For PostgreSQL: "postgresql://user:password@postgresserver/db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Models & Schemas Pattern

```python
# database.py - SQLAlchemy model
from sqlalchemy import Boolean, Column, Integer, String

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)

# schemas.py - Pydantic model
class UserBase(BaseModel):
    email: str

class UserCreate(UserBase):
    password: str

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

    class Config:
        orm_mode = True  # Allows Pydantic to work with ORM objects
```

Database Dependency

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

@app.post("/users/", response_model=schemas.User)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):
    db_user = models.User(email=user.email, hashed_password=user.password)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user
```

Authentication & Security

OAuth2 with Password Flow

```python
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from passlib.context import CryptContext
from jose import JWTError, jwt
from datetime import datetime, timedelta

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

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

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)

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

async def get_current_user(token: str = Depends(oauth2_scheme)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception
    return username

@app.post("/token")
async def login(form_data: OAuth2PasswordRequestForm = Depends()):
    user = authenticate_user(form_data.username, form_data.password)
    if not user:
        raise HTTPException(status_code=400, detail="Incorrect username or
```

```
password")
    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"}

@app.get("/users/me")
async def read_users_me(current_user: str = Depends(get_current_user)):
    return {"username": current_user}
```

Middleware & CORS

CORS Configuration

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],  # React app origin
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Custom Middleware

```
from starlette.middleware.base import BaseHTTPMiddleware
import time

class TimingMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        start_time = time.time()
        response = await call_next(request)
        process_time = time.time() - start_time
        response.headers["X-Process-Time"] = str(process_time)
        return response

app.add_middleware(TimingMiddleware)
```

Exception Handling

Custom Exception Handlers

```
from fastapi import HTTPException, Request
from fastapi.responses import JSONResponse

class UnicornException(Exception):
    def __init__(self, name: str):
        self.name = name

@app.exception_handler(UnicornException)
async def unicorn_exception_handler(request: Request, exc:
UnicornException):
    return JSONResponse(
        status_code=418,
        content={"message": f"Oops! {exc.name} did something."},
    )

@app.get("/unicorns/{name}")
async def read_unicorn(name: str):
    if name == "yolo":
        raise UnicornException(name=name)
    return {"unicorn_name": name}
```

Override Default Handlers

```
from fastapi.exceptions import RequestValidationError
from fastapi.responses import PlainTextResponse

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request, exc):
    return PlainTextResponse(str(exc), status_code=400)
```

Background Tasks

Simple Background Task

```
from fastapi import BackgroundTasks

def write_log(message: str):
    with open("log.txt", mode="a") as log:
        log.write(message)

@app.post("/send-notification/{email}")
async def send_notification(email: str, background_tasks: BackgroundTasks):
```

```
    background_tasks.add_task(write_log, f"Notification sent to {email}\n")
    return {"message": "Notification sent"}
```

Multiple Background Tasks

```
@app.post("/send-notification/{email}")
async def send_notification(
    email: str,
    background_tasks: BackgroundTasks
):
    background_tasks.add_task(write_log, f"Notification sent to {email}\n")
    background_tasks.add_task(send_email, email, message="Some
notification")
    return {"message": "Notification sent"}
```

Testing

Basic Test Setup

```
from fastapi.testclient import TestClient

client = TestClient(app)

def test_read_main():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello World"}

def test_create_item():
    response = client.post(
        "/items/",
        json={"name": "Foo", "price": 42.0}
    )
    assert response.status_code == 200
    assert response.json()["name"] == "Foo"
```

Testing with Database Override

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"

engine = create_engine(SQLALCHEMY_DATABASE_URL)
```

```
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)

def override_get_db():
    try:
        db = TestingSessionLocal()
        yield db
    finally:
        db.close()

app.dependency_overrides[get_db] = override_get_db

def test_create_user():
    response = client.post(
        "/users/",
        json={"email": "test@example.com", "password": "secret"}
    )
    assert response.status_code == 200
```

Advanced Features

Response Model & Status Codes

```
from fastapi import status

@app.post("/items/", response_model=Item,
status_code=status.HTTP_201_CREATED)
async def create_item(item: Item):
    return item

@app.get("/items/{item_id}", response_model=Item,
         responses={404: {"description": "Item not found"}})
async def read_item(item_id: int):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found")
    return items[item_id]
```

File Upload

```
from fastapi import File, UploadFile

@app.post("/files/")
async def create_file(file: bytes = File()):
    return {"file_size": len(file)}
```

```python
@app.post("/uploadfile/")
async def create_upload_file(file: UploadFile = File()):
    contents = await file.read()
    return {"filename": file.filename, "size": len(contents)}
```

Form Data

```python
from fastapi import Form

@app.post("/login/")
async def login(username: str = Form(), password: str = Form()):
    return {"username": username}
```

WebSocket Support

```python
from fastapi import WebSocket

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
        await websocket.send_text(f"Message text was: {data}")
```

Alembic Migrations

Initial Setup

```bash
alembic init alembic
```

Configuration (alembic/env.py)

```python
from models import Base
target_metadata = Base.metadata

def run_migrations_online():
    configuration = config.get_section(config.config_ini_section)
    configuration["sqlalchemy.url"] = SQLALCHEMY_DATABASE_URL
    # ... rest of migration logic
```

Create Migration

```
alembic revision --autogenerate -m "Add user table"
alembic upgrade head
alembic downgrade -1
```

Performance Optimization

Enable Gzip Compression

```
from fastapi.middleware.gzip import GZipMiddleware

app.add_middleware(GZipMiddleware, minimum_size=1000)
```

Response Caching Strategy

```
from functools import lru_cache

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

@app.get("/settings")
async def read_settings(settings: Settings = Depends(get_settings)):
    return settings
```

Async Database Operations

```
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine

engine =
create_async_engine("postgresql+asyncpg://user:password@localhost/db")

async def get_async_db():
    async with AsyncSession(engine) as session:
        yield session

@app.get("/users/")
async def read_users(db: AsyncSession = Depends(get_async_db)):
    result = await db.execute(select(User))
    return result.scalars().all()
```

Deployment Considerations

Production Server (Uvicorn)

```
# Development
uvicorn main:app --reload

# Production
uvicorn main:app --host 0.0.0.0 --port 8000 --workers 4
```

Docker Configuration

```
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Environment Variables

```
from pydantic import BaseSettings

class Settings(BaseSettings):
    database_url: str
    secret_key: str

    class Config:
        env_file = ".env"

settings = Settings()
```

Router Organization

APIRouter for Modular Structure

```
# routers/items.py
from fastapi import APIRouter

router = APIRouter(
    prefix="/items",
    tags=["items"],
    responses={404: {"description": "Not found"}},
```

```
)

@router.get("/")
async def read_items():
    return [{"name": "Item Foo"}]

@router.get("/{item_id}")
async def read_item(item_id: int):
    return {"name": "Fake Item", "item_id": item_id}

# main.py
from routers import items

app.include_router(items.router)
```

Key Architectural Patterns

1. **Separation of concerns**: models.py (SQLAlchemy), schemas.py (Pydantic), routers (endpoints)
2. **Dependency injection**: Database sessions, authentication, common parameters
3. **Type-driven validation**: Pydantic models ensure data integrity at API boundaries
4. **Async-first design**: Native async/await for I/O operations
5. **Middleware stack**: CORS, authentication, logging, error handling
6. **Testing isolation**: Override dependencies for unit/integration tests

---

Common Pitfalls & Solutions

Problem: `orm_mode` not set in Pydantic model
Solution: Add `class Config: orm_mode = True` to convert SQLAlchemy models

Problem: Circular imports between routers
Solution: Use forward references with `from __future__ import annotations`

Problem: Database connection leaks
Solution: Always use dependency injection with try/finally for session cleanup

Problem: JWT token expiration not handled
Solution: Implement token refresh mechanism and proper exception handling

Problem: Large file uploads blocking event loop
Solution: Use `UploadFile` with streaming and background tasks for processing