🚀 Complete FastAPI Production Guide: Deployment, Debugging, Testing & Logging

📋 Table of Contents

🌐 Deployment

📁 Production-Ready Project Structure

```
fastapi-app/
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── api/
│   │   ├── __init__.py
│   │   └── v1/
│   │       ├── __init__.py
│   │       ├── auth.py
│   │       └── users.py
│   ├── core/
│   │   ├── __init__.py
│   │   ├── config.py
│   │   └── security.py
│   ├── models/
│   │   ├── __init__.py
│   │   └── user.py
│   └── services/
│       ├── __init__.py
│       └── user_service.py
├── tests/
├── requirements.txt
├── Dockerfile
├── docker-compose.yml
├── gunicorn.conf.py
├── nginx.conf
└── .env
```

⚙️ Production Configuration

Environment Configuration

```python
# app/core/config.py
from pydantic import BaseSettings
from typing import Optional

class Settings(BaseSettings):
    app_name: str = "FastAPI App"
    debug: bool = False
    secret_key: str
    database_url: str
    redis_url: Optional[str] = None

    # Security
    access_token_expire_minutes: int = 30
    refresh_token_expire_days: int = 7

    # CORS
    allowed_hosts: list = ["*"]

    class Config:
        env_file = ".env"

settings = Settings()
```

Main Application with Production Settings

```python
# app/main.py
from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware
from fastapi.middleware.trustedhost import TrustedHostMiddleware
from contextlib import asynccontextmanager
import logging
from app.core.config import settings
from app.api.v1 import auth, users

# Logger setup
logger = logging.getLogger(__name__)

@asynccontextmanager
async def lifespan(app: FastAPI):
    """Application startup and shutdown"""
    # Startup
    logger.info("🚀 Starting FastAPI application...")
    yield
    # Shutdown
    logger.info("🔄 Shutting down FastAPI application...")

app = FastAPI(
```

```
    title=settings.app_name,
    description="Production FastAPI Application",
    version="1.0.0",
    debug=settings.debug,
    lifespan=lifespan,
    docs_url="/docs" if settings.debug else None,  # Disable docs in production
    redoc_url="/redoc" if settings.debug else None
)

# Security Middleware
app.add_middleware(
    TrustedHostMiddleware,
    allowed_hosts=settings.allowed_hosts
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Configure properly for production
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Health Check Endpoint
@app.get("/health", tags=["Health"])
async def health_check():
    """🏥 Health check endpoint for load balancers"""
    return {"status": "healthy", "version": "1.0.0"}

# Include Routers
app.include_router(auth.router, prefix="/api/v1", tags=["Auth"])
app.include_router(users.router, prefix="/api/v1", tags=["Users"])
```

🐳 Docker Configuration

Dockerfile

```
FROM python:3.11-slim

# Set environment variables
ENV PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1 \
    PIP_NO_CACHE_DIR=1 \
    PIP_DISABLE_PIP_VERSION_CHECK=1

# Create app directory
WORKDIR /app
```

```
# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    && rm -rf /var/lib/apt/lists/*

# Copy and install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Create non-root user
RUN useradd --create-home --shell /bin/bash app && chown -R app:app /app
USER app

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:8000/health || exit 1

# Expose port
EXPOSE 8000

# Run application
CMD ["gunicorn", "app.main:app", "-c", "gunicorn.conf.py"]
```

Docker Compose

```
# docker-compose.yml
version: '3.8'

services:
  app:
    build: .
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://user:password@db:5432/fastapi_db
      - REDIS_URL=redis://redis:6379
    depends_on:
      - db
      - redis
    volumes:
      - ./logs:/app/logs

  db:
    image: postgres:15
    environment:
```

```
      POSTGRES_DB: fastapi_db
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - app

volumes:
  postgres_data:
```

🌐 Production Server Configuration

Gunicorn Configuration

```
# gunicorn.conf.py
import multiprocessing

# Server socket
bind = "0.0.0.0:8000"
backlog = 2048

# Worker processes
workers = multiprocessing.cpu_count() * 2 + 1
worker_class = "uvicorn.workers.UvicornWorker"
worker_connections = 1000
max_requests = 1000
max_requests_jitter = 100

# Timeout
timeout = 30
keepalive = 2
```

```
# Logging
accesslog = "/app/logs/access.log"
errorlog = "/app/logs/error.log"
loglevel = "info"
access_log_format = '%(h)s %(l)s %(u)s %(t)s "%(r)s" %(s)s %(b)s "%(f)s" "%(a)s" %
(D)s'

# Process naming
proc_name = "fastapi-app"

# Server mechanics
preload_app = True
daemon = False
pidfile = "/app/gunicorn.pid"
user = "app"
group = "app"

# SSL (if needed)
# keyfile = "/path/to/keyfile"
# certfile = "/path/to/certfile"
```

Nginx Configuration

```
# nginx.conf
events {
    worker_connections 1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    # Logging
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;

    # Gzip compression
    gzip on;
    gzip_vary on;
    gzip_min_length 1024;
    gzip_types text/plain text/css text/xml text/javascript
               application/json application/javascript
               application/xml+rss application/atom+xml;

    # Rate limiting
    limit_req_zone $binary_remote_addr zone=api:10m rate=10r/s;

    upstream fastapi_backend {
```

```
        server app:8000;
        # Add more servers for load balancing
        # server app2:8000;
    }

    server {
        listen 80;
        server_name yourdomain.com;

        # Rate limiting
        limit_req zone=api burst=20 nodelay;

        # Security headers
        add_header X-Content-Type-Options nosniff;
        add_header X-Frame-Options DENY;
        add_header X-XSS-Protection "1; mode=block";

        # Proxy settings
        location / {
            proxy_pass http://fastapi_backend;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # Static files (if any)
        location /static/ {
            alias /app/static/;
            expires 30d;
            add_header Cache-Control "public, immutable";
        }
    }
}
```

Cloud Deployment Examples

Heroku Deployment

```
# Procfile
web: gunicorn app.main:app -c gunicorn.conf.py

# runtime.txt
python-3.11.0
```

Railway Deployment

```
// railway.json
{
  "build": {
    "builder": "DOCKERFILE"
  },
  "deploy": {
    "startCommand": "gunicorn app.main:app -c gunicorn.conf.py",
    "healthcheckPath": "/health"
  }
}
```

🐛 Debugging

🔧 Debug Mode Configuration

Enable Debug Mode

```
# app/main.py
import os
from fastapi import FastAPI
import logging

# Configure debug mode
DEBUG = os.getenv("DEBUG", "false").lower() == "true"

app = FastAPI(
    debug=DEBUG,
    title="FastAPI Debug App"
)

if DEBUG:
    logging.basicConfig(level=logging.DEBUG)
    logger = logging.getLogger(__name__)
    logger.debug("🐛 Debug mode enabled!")

@app.get("/debug-info")
async def debug_info():
    """Debug endpoint - only available in debug mode"""
    if not DEBUG:
        return {"message": "Debug info not available in production"}

    import sys
    import platform

    return {
        "python_version": sys.version,
        "platform": platform.platform(),
```

```
        "debug_mode": DEBUG,
        "environment": os.environ.get("ENVIRONMENT", "development")
    }
```

🔍 Interactive Debugging

Using PDB (Python Debugger)

```
import pdb

@app.get("/users/{user_id}")
async def get_user(user_id: int):
    # Set breakpoint for debugging
    pdb.set_trace()  # 🛑 Execution will pause here

    user = await get_user_from_db(user_id)
    return user
```

Using Debugpy for Remote Debugging

```
# Remote debugging setup
import debugpy
import os

if os.getenv("ENABLE_DEBUGPY", "false").lower() == "true":
    debugpy.listen(("0.0.0.0", 5678))
    print("⚡ Waiting for debugger to attach...")
    debugpy.wait_for_client()

@app.get("/")
async def root():
    # Set breakpoint here in your IDE
    message = "Hello Debug World!"
    return {"message": message}
```

📝 Advanced Debugging with Middleware

Request/Response Debug Middleware

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

```python
logger = logging.getLogger(__name__)

class DebugMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        start_time = time.time()

        # Log request details
        logger.debug(f"🔵 Request: {request.method} {request.url}")
        logger.debug(f"🔵 Headers: {dict(request.headers)}")

        # Get request body for debugging
        if request.method in ["POST", "PUT", "PATCH"]:
            body = await request.body()
            logger.debug(f"🔵 Body: {body.decode() if body else 'Empty'}")

        response = await call_next(request)

        process_time = time.time() - start_time
        logger.debug(f"🔵 Response: {response.status_code} - {process_time:.4f}s")

        return response

# Add middleware only in debug mode
if DEBUG:
    app.add_middleware(DebugMiddleware)
```

🚨 Exception Handling & Error Tracking

Custom Exception Handler

```python
from fastapi import Request, HTTPException
from fastapi.responses import JSONResponse
import traceback
import logging

logger = logging.getLogger(__name__)

@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    """🚨 Global exception handler for debugging"""

    # Log the full traceback
    logger.error(f"❌ Unhandled exception: {exc}")
    logger.error(f"❌ Traceback: {traceback.format_exc()}")
    logger.error(f"❌ Request: {request.method} {request.url}")

    if DEBUG:
```

```
        # Return detailed error in debug mode
        return JSONResponse(
            status_code=500,
            content={
                "error": str(exc),
                "type": type(exc).__name__,
                "traceback": traceback.format_exc().split("\n")
            }
        )
    else:
        # Generic error in production
        return JSONResponse(
            status_code=500,
            content={"error": "Internal server error"}
        )

@app.exception_handler(HTTPException)
async def http_exception_handler(request: Request, exc: HTTPException):
    """🚨 HTTP exception handler"""
    logger.warning(f"⚠️ HTTP Exception: {exc.status_code} - {exc.detail}")
    logger.warning(f"⚠️ Request: {request.method} {request.url}")

    return JSONResponse(
        status_code=exc.status_code,
        content={"error": exc.detail, "status_code": exc.status_code}
    )
```

🔍 Debugging Async Code

Async Debugging Tools

```
import asyncio
import aiomonitor

# Monitor async tasks
@app.on_event("startup")
async def start_monitoring():
    if DEBUG:
        # Start async monitor
        aiomonitor.start_monitor(loop=asyncio.get_event_loop())
        logger.debug("🔍 Async monitor started")

@app.get("/async-debug")
async def async_debug_example():
    """Example of debugging async operations"""
    logger.debug("🔄 Starting async operation")

    # Simulate async work
```

```
    await asyncio.sleep(1)

    # Check current running tasks
    current_task = asyncio.current_task()
    all_tasks = asyncio.all_tasks()

    logger.debug(f"🔄 Current task: {current_task}")
    logger.debug(f"🔄 Total tasks: {len(all_tasks)}")

    return {
        "current_task": str(current_task),
        "total_tasks": len(all_tasks),
        "task_names": [task.get_name() for task in all_tasks]
    }
```

📊 Performance Debugging

Performance Profiling Middleware

```
import cProfile
import pstats
from io import StringIO

class ProfilingMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        if not DEBUG:
            return await call_next(request)

        # Start profiling
        profiler = cProfile.Profile()
        profiler.enable()

        response = await call_next(request)

        # Stop profiling
        profiler.disable()

        # Get stats
        stats_stream = StringIO()
        stats = pstats.Stats(profiler, stream=stats_stream)
        stats.sort_stats('cumulative')
        stats.print_stats(20)  # Top 20 functions

        # Log performance stats
        logger.debug(f"📊 Performance Stats for
{request.url}:\n{stats_stream.getvalue()}")

        return response
```

```
if DEBUG:
    app.add_middleware(ProfilingMiddleware)
```

🧪 Testing

🧬 Testing Setup & Structure

Test Configuration

```
# tests/conftest.py
import pytest
from fastapi.testclient import TestClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
import os
import asyncio
from httpx import AsyncClient

from app.main import app
from app.database import Base, get_db

# Test database
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread":
False})
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

def override_get_db():
    """Override database dependency for testing"""
    try:
        db = TestingSessionLocal()
        yield db
    finally:
        db.close()

app.dependency_overrides[get_db] = override_get_db

@pytest.fixture(scope="session")
def event_loop():
    """Create event loop for async tests"""
    loop = asyncio.get_event_loop_policy().new_event_loop()
    yield loop
    loop.close()

@pytest.fixture
def client():
    """Test client fixture"""
```

```
    with TestClient(app) as test_client:
        yield test_client

@pytest.fixture
async def async_client():
    """Async test client fixture"""
    async with AsyncClient(app=app, base_url="http://test") as ac:
        yield ac

@pytest.fixture(autouse=True)
def create_test_database():
    """Create test database before each test"""
    Base.metadata.create_all(bind=engine)
    yield
    Base.metadata.drop_all(bind=engine)
```

🧪 Unit Testing

Basic Unit Tests

```
# tests/test_main.py
import pytest
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_health_check():
    """🏥 Test health check endpoint"""
    response = client.get("/health")
    assert response.status_code == 200
    assert response.json() == {"status": "healthy", "version": "1.0.0"}

def test_root_endpoint():
    """🏠 Test root endpoint"""
    response = client.get("/")
    assert response.status_code == 200
    assert "message" in response.json()

class TestUserEndpoints:
    """👥 User endpoint tests"""

    def test_create_user(self, client):
        """✨ Test user creation"""
        user_data = {
            "username": "testuser",
            "email": "test@example.com",
            "password": "TestPass123",
```

```
            "full_name": "Test User"
        }

        response = client.post("/api/v1/users/register", json=user_data)

        assert response.status_code == 201
        data = response.json()
        assert data["success"] is True
        assert data["user"]["username"] == user_data["username"]
        assert data["user"]["email"] == user_data["email"]

    def test_create_user_duplicate_email(self, client):
        """❌ Test duplicate email validation"""
        user_data = {
            "username": "testuser",
            "email": "test@example.com",
            "password": "TestPass123",
            "full_name": "Test User"
        }

        # Create first user
        client.post("/api/v1/users/register", json=user_data)

        # Try to create second user with same email
        user_data["username"] = "testuser2"
        response = client.post("/api/v1/users/register", json=user_data)

        assert response.status_code == 400
        assert "already registered" in response.json()["detail"]

    def test_login_user(self, client):
        """🔐 Test user login"""
        # First create a user
        user_data = {
            "username": "testuser",
            "email": "test@example.com",
            "password": "TestPass123",
            "full_name": "Test User"
        }
        client.post("/api/v1/users/register", json=user_data)

        # Then login
        login_data = {
            "email": "test@example.com",
            "password": "TestPass123"
        }

        response = client.post("/api/v1/auth/login", json=login_data)

        assert response.status_code == 200
        data = response.json()
```

```
        assert "access_token" in data
        assert "refresh_token" in data
        assert data["token_type"] == "bearer"
```

🔄 Async Testing

Async Test Examples

```
# tests/test_async_endpoints.py
import pytest
import asyncio
from httpx import AsyncClient
from app.main import app

@pytest.mark.asyncio
async def test_async_endpoint():
    """🔄 Test async endpoint"""
    async with AsyncClient(app=app, base_url="http://test") as ac:
        response = await ac.get("/async-endpoint")
        assert response.status_code == 200

@pytest.mark.asyncio
async def test_concurrent_requests():
    """⚡ Test concurrent requests"""
    async with AsyncClient(app=app, base_url="http://test") as ac:
        # Create multiple concurrent requests
        tasks = [
            ac.get(f"/users/{i}")
            for i in range(1, 11)
        ]

        responses = await asyncio.gather(*tasks, return_exceptions=True)

        # Verify all requests completed
        assert len(responses) == 10
        successful_responses = [r for r in responses if not isinstance(r,
Exception)]
        assert len(successful_responses) > 0
```

🧪 Integration Testing

Database Integration Tests

```
# tests/test_database_integration.py
import pytest
from app.models.user import UserInDB
```

```python
from app.services.user_service import UserService

@pytest.mark.asyncio
async def test_user_crud_operations():
    """📊 Test complete CRUD operations"""
    user_service = UserService()

    # Create user
    user_data = {
        "username": "testuser",
        "email": "test@example.com",
        "password": "TestPass123",
        "full_name": "Test User"
    }

    # Test Create
    created_user = await user_service.create_user(user_data)
    assert created_user.username == user_data["username"]
    assert created_user.email == user_data["email"]

    # Test Read
    retrieved_user = await user_service.get_user_by_email(user_data["email"])
    assert retrieved_user is not None
    assert retrieved_user.id == created_user.id

    # Test Update
    update_data = {"full_name": "Updated User"}
    updated_user = await user_service.update_user(created_user.id, update_data)
    assert updated_user.full_name == "Updated User"

    # Test Delete
    deleted = await user_service.delete_user(created_user.id)
    assert deleted is True

    # Verify deletion
    deleted_user = await user_service.get_user_by_id(created_user.id)
    assert deleted_user is None
```

🔒 Security Testing

Authentication & Authorization Tests

```python
# tests/test_security.py
import jwt
import pytest
from app.core.security import create_access_token, verify_token

class TestAuthentication:
```

```python
    """🔒 Authentication tests"""

    def test_create_access_token(self):
        """🎫 Test token creation"""
        user_data = {"sub": "user123", "email": "test@example.com"}
        token = create_access_token(user_data)

        assert isinstance(token, str)
        assert len(token) > 0

        # Decode and verify
        decoded = jwt.decode(token, options={"verify_signature": False})
        assert decoded["sub"] == "user123"
        assert decoded["email"] == "test@example.com"

    def test_protected_endpoint_without_token(self, client):
        """🚫 Test protected endpoint without token"""
        response = client.get("/api/v1/users/me")
        assert response.status_code == 401

    def test_protected_endpoint_with_invalid_token(self, client):
        """❌ Test protected endpoint with invalid token"""
        headers = {"Authorization": "Bearer invalid_token"}
        response = client.get("/api/v1/users/me", headers=headers)
        assert response.status_code == 401

    def test_protected_endpoint_with_valid_token(self, client):
        """✅ Test protected endpoint with valid token"""
        # First create and login user
        user_data = {
            "username": "testuser",
            "email": "test@example.com",
            "password": "TestPass123",
            "full_name": "Test User"
        }
        client.post("/api/v1/users/register", json=user_data)

        login_response = client.post("/api/v1/auth/login", json={
            "email": "test@example.com",
            "password": "TestPass123"
        })

        token = login_response.json()["access_token"]
        headers = {"Authorization": f"Bearer {token}"}

        response = client.get("/api/v1/users/me", headers=headers)
        assert response.status_code == 200
        assert response.json()["user"]["email"] == "test@example.com"
```

🧪 Test Coverage & Reporting

Pytest Configuration

```
# pytest.ini
[tool:pytest]
testpaths = tests
python_files = test_*.py
python_classes = Test*
python_functions = test_*
addopts =
    --verbose
    --tb=short
    --cov=app
    --cov-report=html
    --cov-report=term-missing
    --cov-fail-under=80
    -p no:warnings

markers =
    unit: Unit tests
    integration: Integration tests
    slow: Slow tests
    security: Security tests
```

Coverage Configuration

```
# .coveragerc
[run]
source = app
omit =
    */tests/*
    */venv/*
    */__pycache__/*
    */migrations/*

[report]
exclude_lines =
    pragma: no cover
    def __repr__
    raise AssertionError
    raise NotImplementedError
```

🚀 Performance Testing

Load Testing with Locust

```python
# tests/load_test.py
from locust import HttpUser, task, between

class FastAPIUser(HttpUser):
    wait_time = between(1, 3)

    def on_start(self):
        """Login and get token"""
        response = self.client.post("/api/v1/auth/login", json={
            "email": "test@example.com",
            "password": "TestPass123"
        })

        if response.status_code == 200:
            self.token = response.json()["access_token"]
            self.headers = {"Authorization": f"Bearer {self.token}"}

    @task(3)
    def get_users(self):
        """🏃 Load test user listing"""
        self.client.get("/api/v1/users/", headers=self.headers)

    @task(2)
    def get_user_profile(self):
        """👤 Load test user profile"""
        self.client.get("/api/v1/users/me", headers=self.headers)

    @task(1)
    def update_user(self):
        """📝 Load test user update"""
        self.client.put("/api/v1/users/update-account",
                        json={"full_name": "Load Test User"},
                        headers=self.headers)
```

📊 Logging

📝 Basic Logging Setup

Structured Logging Configuration

```python
# app/core/logging.py
import logging
from logging.config import dictConfig
import os
import sys
from typing import Dict, Any
```

```
def setup_logging() -> Dict[str, Any]:
    """🔧 Configure logging for the application"""

    log_level = os.getenv("LOG_LEVEL", "INFO").upper()

    logging_config = {
        "version": 1,
        "disable_existing_loggers": False,
        "formatters": {
            "default": {
                "format": "%(asctime)s | %(levelname)-8s | %(name)s:%(lineno)d | %
(message)s",
                "datefmt": "%Y-%m-%d %H:%M:%S",
            },
            "json": {
                "format": '{"timestamp": "%(asctime)s", "level": "%(levelname)s",
"logger": "%(name)s", "line": %(lineno)d, "message": "%(message)s"}',
                "datefmt": "%Y-%m-%d %H:%M:%S",
            },
            "detailed": {
                "format": "%(asctime)s | %(levelname)-8s | %(name)s:%(lineno)d | %
(funcName)s() | %(message)s",
                "datefmt": "%Y-%m-%d %H:%M:%S",
            }
        },
        "handlers": {
            "console": {
                "class": "logging.StreamHandler",
                "level": log_level,
                "formatter": "default",
                "stream": sys.stdout,
            },
            "file": {
                "class": "logging.handlers.RotatingFileHandler",
                "level": log_level,
                "formatter": "detailed",
                "filename": "logs/app.log",
                "maxBytes": 10485760,  # 10MB
                "backupCount": 5,
                "encoding": "utf8",
            },
            "error_file": {
                "class": "logging.handlers.RotatingFileHandler",
                "level": "ERROR",
                "formatter": "json",
                "filename": "logs/errors.log",
                "maxBytes": 10485760,  # 10MB
                "backupCount": 5,
                "encoding": "utf8",
            },
        },
```

```
        "loggers": {
            "": {  # Root logger
                "handlers": ["console", "file"],
                "level": log_level,
                "propagate": False,
            },
            "uvicorn": {
                "handlers": ["console"],
                "level": "INFO",
                "propagate": False,
            },
            "uvicorn.access": {
                "handlers": ["file"],
                "level": "INFO",
                "propagate": False,
            },
            "app": {
                "handlers": ["console", "file", "error_file"],
                "level": log_level,
                "propagate": False,
            },
        },
    }

    # Create logs directory if it doesn't exist
    os.makedirs("logs", exist_ok=True)

    dictConfig(logging_config)
    return logging_config

# Initialize logging
logger = logging.getLogger("app")
```

📊 Request/Response Logging Middleware

Advanced Logging Middleware

```
# app/middleware/logging.py
from starlette.middleware.base import BaseHTTPMiddleware
from starlette.requests import Request
import logging
import time
import json
import uuid

logger = logging.getLogger("app.middleware")

class LoggingMiddleware(BaseHTTPMiddleware):
```

```
    async def dispatch(self, request: Request, call_next):
        # Generate request ID for tracing
        request_id = str(uuid.uuid4())
        start_time = time.time()

        # Log request
        await self._log_request(request, request_id)

        # Process request
        response = await call_next(request)

        # Calculate processing time
        process_time = time.time() - start_time

        # Log response
        self._log_response(response, process_time, request_id)

        # Add request ID to response headers
        response.headers["X-Request-ID"] = request_id

        return response

    async def _log_request(self, request: Request, request_id: str):
        """📥 Log incoming request"""

        # Get client IP
        client_ip = request.client.host if request.client else "unknown"

        # Get request body for POST/PUT/PATCH
        body = None
        if request.method in ["POST", "PUT", "PATCH"]:
            body = await request.body()
            body_str = body.decode() if body else ""

            # Don't log sensitive data
            if any(sensitive in request.url.path for sensitive in ["/auth/",
"/login", "/register"]):
                body_str = "[REDACTED]"

        logger.info(
            "📥 Request",
            extra={
                "request_id": request_id,
                "method": request.method,
                "url": str(request.url),
                "path": request.url.path,
                "client_ip": client_ip,
                "user_agent": request.headers.get("user-agent", ""),
                "body": body_str if body else None,
                "headers": dict(request.headers)
            }
```

```
        )

    def _log_response(self, response, process_time: float, request_id: str):
        """📤 Log outgoing response"""

        log_level = logging.INFO
        if response.status_code >= 400:
            log_level = logging.WARNING
        if response.status_code >= 500:
            log_level = logging.ERROR

        logger.log(
            log_level,
            f"📤 Response - {response.status_code}",
            extra={
                "request_id": request_id,
                "status_code": response.status_code,
                "process_time": round(process_time, 4),
                "response_headers": dict(response.headers)
            }
        )
```

🎯 Business Logic Logging

Service Layer Logging

```
# app/services/user_service.py
import logging
from typing import Optional, Dict, Any
from app.models.internal.user import UserInDB

logger = logging.getLogger("app.services.user")

class UserService:

    async def register_user(self, user_data: Dict[str, Any]) -> UserInDB:
        """👤 Register new user with comprehensive logging"""

        logger.info(
            f"🔄 Starting user registration",
            extra={
                "action": "register_user",
                "username": user_data.get("username"),
                "email": user_data.get("email")
            }
        )

        try:
```

```
            # Check if user exists
            existing_user = await self.get_user_by_email(user_data["email"])
            if existing_user:
                logger.warning(
                    f"❌ Registration failed - email already exists",
                    extra={
                        "action": "register_user",
                        "email": user_data["email"],
                        "reason": "email_exists"
                    }
                )
                raise ValueError("Email already registered")

            # Create user
            user = UserInDB(**user_data)
            saved_user = await self.save_user_to_db(user)

            logger.info(
                f"✅ User registered successfully",
                extra={
                    "action": "register_user",
                    "user_id": str(saved_user.id),
                    "username": saved_user.username,
                    "email": saved_user.email
                }
            )

            return saved_user

        except Exception as e:
            logger.error(
                f"💥 User registration failed",
                extra={
                    "action": "register_user",
                    "email": user_data.get("email"),
                    "error": str(e),
                    "error_type": type(e).__name__
                },
                exc_info=True
            )
            raise

    async def login_user(self, email: str, password: str) -> Optional[UserInDB]:
        """🔐 User login with security logging"""

        logger.info(
            f"🔑 Login attempt",
            extra={
                "action": "login_user",
                "email": email
            }
```

```
        )

        try:
            user = await self.get_user_by_email(email)
            if not user:
                logger.warning(
                    f"🚫 Login failed - user not found",
                    extra={
                        "action": "login_user",
                        "email": email,
                        "reason": "user_not_found"
                    }
                )
                return None

            if not self.verify_password(password, user.hashed_password):
                logger.warning(
                    f"🚫 Login failed - invalid password",
                    extra={
                        "action": "login_user",
                        "email": email,
                        "user_id": str(user.id),
                        "reason": "invalid_password"
                    }
                )
                return None

            # Update last login
            user.last_login = datetime.utcnow()
            await self.update_user_in_db(user)

            logger.info(
                f"✅ Login successful",
                extra={
                    "action": "login_user",
                    "user_id": str(user.id),
                    "email": email,
                    "last_login": user.last_login.isoformat()
                }
            )

            return user

        except Exception as e:
            logger.error(
                f"💥 Login error",
                extra={
                    "action": "login_user",
                    "email": email,
                    "error": str(e),
                    "error_type": type(e).__name__
```

```
                },
                exc_info=True
            )
            raise
```

🔍 Error and Exception Logging

Comprehensive Error Logging

```python
# app/core/exception_handlers.py
import logging
import traceback
from fastapi import Request, HTTPException
from fastapi.responses import JSONResponse
import sys

logger = logging.getLogger("app.exceptions")

async def log_and_handle_exception(request: Request, exc: Exception):
    """🚨 Comprehensive exception logging and handling"""

    # Extract request information
    request_info = {
        "method": request.method,
        "url": str(request.url),
        "client_ip": request.client.host if request.client else "unknown",
        "user_agent": request.headers.get("user-agent", ""),
        "request_id": request.headers.get("X-Request-ID", "")
    }

    # Get exception details
    exc_type = type(exc).__name__
    exc_message = str(exc)
    exc_traceback = traceback.format_exception(type(exc), exc, exc.__traceback__)

    if isinstance(exc, HTTPException):
        # HTTP exceptions (4xx, 5xx)
        log_level = logging.WARNING if exc.status_code < 500 else logging.ERROR

        logger.log(
            log_level,
            f"🚨 HTTP Exception: {exc.status_code} - {exc.detail}",
            extra={
                "exception_type": "HTTPException",
                "status_code": exc.status_code,
                "detail": exc.detail,
                **request_info
            }
```

```
        )

        return JSONResponse(
            status_code=exc.status_code,
            content={
                "error": exc.detail,
                "status_code": exc.status_code,
                "request_id": request_info["request_id"]
            }
        )

    else:
        # Unhandled exceptions
        logger.error(
            f"💥 Unhandled Exception: {exc_type} - {exc_message}",
            extra={
                "exception_type": exc_type,
                "exception_message": exc_message,
                "traceback": exc_traceback,
                **request_info
            },
            exc_info=True
        )

        # Send to error tracking service (Sentry, etc.)
        # sentry_sdk.capture_exception(exc)

        return JSONResponse(
            status_code=500,
            content={
                "error": "Internal server error",
                "request_id": request_info["request_id"]
            }
        )
```

📊 Performance and Monitoring Logs

Performance Monitoring

```
# app/middleware/performance.py
import logging
import time
import psutil
import asyncio
from starlette.middleware.base import BaseHTTPMiddleware

logger = logging.getLogger("app.performance")
```

```
class PerformanceMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        start_time = time.time()
        start_memory = psutil.Process().memory_info().rss / 1024 / 1024  # MB

        response = await call_next(request)

        end_time = time.time()
        end_memory = psutil.Process().memory_info().rss / 1024 / 1024  # MB

        process_time = end_time - start_time
        memory_diff = end_memory - start_memory

        # Log slow requests
        if process_time > 1.0:  # Requests taking more than 1 second
            logger.warning(
                f"🐌 Slow request detected",
                extra={
                    "url": str(request.url),
                    "method": request.method,
                    "process_time": round(process_time, 4),
                    "memory_usage": round(memory_diff, 2),
                    "status_code": response.status_code
                }
            )

        # Log performance metrics
        logger.info(
            f"📊 Performance metrics",
            extra={
                "url": str(request.url),
                "method": request.method,
                "process_time": round(process_time, 4),
                "memory_usage": round(memory_diff, 2),
                "memory_total": round(end_memory, 2),
                "status_code": response.status_code
            }
        )

        return response
```

📈 Structured Logging with Context

Context-Aware Logging

```
# app/core/context_logging.py
import logging
from contextvars import ContextVar
```

```python
from typing import Dict, Any, Optional
import uuid

# Context variables for request tracing
request_id_var: ContextVar[str] = ContextVar('request_id', default='')
user_id_var: ContextVar[str] = ContextVar('user_id', default='')

class ContextFilter(logging.Filter):
    """Add context variables to log records"""

    def filter(self, record):
        record.request_id = request_id_var.get('')
        record.user_id = user_id_var.get('')
        return True

class ContextLogger:
    """Logger with automatic context injection"""

    def __init__(self, name: str):
        self.logger = logging.getLogger(name)
        self.logger.addFilter(ContextFilter())

    def _log_with_context(self, level: int, message: str, extra: Optional[Dict[str,
Any]] = None):
        """Log with automatic context injection"""
        if extra is None:
            extra = {}

        # Add context to extra data
        extra.update({
            'request_id': request_id_var.get(''),
            'user_id': user_id_var.get(''),
        })

        self.logger.log(level, message, extra=extra)

    def debug(self, message: str, **kwargs):
        self._log_with_context(logging.DEBUG, message, kwargs)

    def info(self, message: str, **kwargs):
        self._log_with_context(logging.INFO, message, kwargs)

    def warning(self, message: str, **kwargs):
        self._log_with_context(logging.WARNING, message, kwargs)

    def error(self, message: str, **kwargs):
        self._log_with_context(logging.ERROR, message, kwargs)

# Usage example
def set_request_context(request_id: str, user_id: str = ''):
    """Set context for current request"""
```

```
    request_id_var.set(request_id)
    user_id_var.set(user_id)

# Enhanced logging middleware with context
class ContextLoggingMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        request_id = str(uuid.uuid4())
        user_id = getattr(request.state, 'user_id', '')

        # Set context
        set_request_context(request_id, user_id)

        response = await call_next(request)
        response.headers["X-Request-ID"] = request_id

        return response
```

🔧 Log Analysis and Monitoring

Log Analysis Tools Integration

```
# app/core/log_analysis.py
import logging
import json
from datetime import datetime
from typing import Dict, Any

class JSONFormatter(logging.Formatter):
    """Format logs as JSON for better parsing"""

    def format(self, record):
        log_entry = {
            "timestamp": datetime.utcnow().isoformat() + "Z",
            "level": record.levelname,
            "logger": record.name,
            "message": record.getMessage(),
            "module": record.module,
            "function": record.funcName,
            "line": record.lineno,
        }

        # Add extra fields if present
        if hasattr(record, 'request_id'):
            log_entry['request_id'] = record.request_id
        if hasattr(record, 'user_id'):
            log_entry['user_id'] = record.user_id

        # Add exception info if present
```

```
        if record.exc_info:
            log_entry['exception'] = self.formatException(record.exc_info)

        return json.dumps(log_entry)

# Health check logging
async def log_system_health():
    """💾 Log system health metrics"""
    import psutil

    health_logger = logging.getLogger("app.health")

    health_metrics = {
        "cpu_percent": psutil.cpu_percent(interval=1),
        "memory_percent": psutil.virtual_memory().percent,
        "disk_percent": psutil.disk_usage('/').percent,
        "active_connections": len(psutil.net_connections()),
    }

    health_logger.info(
        "💾 System health metrics",
        extra=health_metrics
    )

# Database query logging
class DatabaseQueryLogger:
    """📊 Log database operations"""

    def __init__(self):
        self.logger = logging.getLogger("app.database")

    def log_query(self, query: str, params: tuple = None, duration: float = 0):
        """Log database query with timing"""

        # Don't log sensitive data
        safe_params = "[REDACTED]" if params and any(
            sensitive in query.lower()
            for sensitive in ["password", "token", "secret"]
        ) else params

        self.logger.info(
            f"🗄️ Database query",
            extra={
                "query": query[:200] + "..." if len(query) > 200 else query,
                "params": safe_params,
                "duration": round(duration, 4),
                "slow_query": duration > 1.0
            }
        )

# Initialize enhanced logging
```

```python
def setup_production_logging():
    """🚀 Setup production-ready logging"""

    # Create logs directory
    import os
    os.makedirs("logs", exist_ok=True)

    # Configure formatters
    json_formatter = JSONFormatter()
    standard_formatter = logging.Formatter(
        "%(asctime)s | %(levelname)-8s | %(name)s | %(message)s"
    )

    # Configure handlers
    handlers = {
        "console": logging.StreamHandler(),
        "file": logging.handlers.RotatingFileHandler(
            "logs/app.log", maxBytes=10485760, backupCount=5
        ),
        "json_file": logging.handlers.RotatingFileHandler(
            "logs/app.json.log", maxBytes=10485760, backupCount=5
        ),
        "error_file": logging.handlers.RotatingFileHandler(
            "logs/error.log", maxBytes=10485760, backupCount=5
        ),
    }

    handlers["console"].setFormatter(standard_formatter)
    handlers["file"].setFormatter(standard_formatter)
    handlers["json_file"].setFormatter(json_formatter)
    handlers["error_file"].setFormatter(json_formatter)

    # Configure root logger
    root_logger = logging.getLogger()
    root_logger.setLevel(logging.INFO)

    for handler in handlers.values():
        root_logger.addHandler(handler)
```

📚 References & Further Reading

Below are the links and resources referenced in this guide for deeper exploration:

🚀 Deployment & Production Readiness

- FastAPI Deployment Concepts – Official Docs [^1]
- FastAPI Deployment Best Practices (GitHub) [^2]
- Deployment Concepts – Official Deep Dive [^3]

- FastAPI Best Practices – DEV.to [^4]
- Deploy FastAPI on Koyeb [^5]
- Deploy FastAPI on Render [^17]
- Production Guide – BlueShoe [^10]

🐞 Debugging

- Debugging FastAPI Apps – LinkedIn Guide [^6]
- FastAPI Debugging Tutorial – YouTube [^13]
- FastAPI Debugging Tips – GetOrchestra.io [^18]
- Unraveling Debugging Mysteries – Mathison Blog [^11]
- FastAPI Debugging – Official Docs [^14]

✅ Testing

- Testing FastAPI – GeeksforGeeks Guide [^7]
- FastAPI Testing Tools & Frameworks – Frugal Testing [^12]
- Unit Testing FastAPI – Apidog [^15]
- FastAPI Testing – Official Docs [^19]

📋 Logging

- Logging Best Practices – LinkedIn [^8]
- BetterStack Logging with FastAPI [^9]
- Configure FastAPI Logging (Local & Prod) – Konfuzio [^16]
- StackOverflow Logging Setup [^20]

⁂

📌 Tips

- Use **separate settings/config files** for `development` and `production`.
- Always **enable structured logging** for easier monitoring.
- Incorporate **Pytest + HTTPX** for writing async test cases.
- Use **Gunicorn + Uvicorn workers** for reliable production deployment.
- Integrate tools like **Sentry**, **Prometheus**, or **ELK stack** for better observability.
