FastAPI Project Workflow

Overview of the FastAPI project structure in covering:

- **Project structure representation**
- **Relationships between files & components**
- **Step-by-step workflow for file creation**
- **Database integration (SQLAlchemy & MongoDB)**
- **Dependency injection, middleware, background tasks**
- **Production-ready best practices**

FastAPI Project Structure Overview

```
fastapi_project/
├── app/
│   ├── main.py                    # FastAPI app initialization
│   ├── core/
│   │   ├── config.py              # Configuration & Environment Variables
│   │   ├── security.py            # Authentication & JWT Handling
│   │   └── middleware.py          # Global Middleware (Logging, CORS, etc.)
│   ├── db/
│   │   ├── base.py                # Base class for SQLAlchemy ORM
│   │   ├── session.py             # SQLAlchemy Database Session
│   │   └── mongo.py               # MongoDB Connection (Motor)
│   ├── models/
│   │   ├── user.py                # SQLAlchemy User Model
│   │   └── item.py                # SQLAlchemy Item Model
│   ├── schemas/
│   │   ├── user.py                # Pydantic Schemas (User Input/Response)
│   │   └── item.py                # Pydantic Schemas (Item Input/Response)
│   ├── services/
│   │   ├── user_service.py        # Business Logic for Users
│   │   └── item_service.py        # Business Logic for Items
│   ├── routers/
│   │   ├── users.py               # User API Endpoints
│   │   └── items.py               # Item API Endpoints
│   ├── ml/
│   │   ├── models.py              # Load AI/ML Models (TensorFlow, PyTorch,
│   │   └── huggingface_api.py     # Hugging Face API Integration
│   ├── worker/
│   │   ├── celery_app.py          # Celery Task Queue Configuration
│   │   └── tasks.py               # Background Task Definitions
```

```
├── tests/
│   ├── test_users.py                # Unit & Integration Tests
│   ├── test_items.py                # Unit & Integration Tests
├── requirements.txt                  # Python Dependencies
├── Dockerfile                        # Containerization
├── docker-compose.yml                # Services Orchestration (DB, Redis, Ce
├── .env                              # Environment Variables
└── README.md                         # Documentation
```

Step-by-Step File Creation Order & Dependencies

This guide explains which files should be created first, dependencies, and logical workflow for a production-ready FastAPI application.

Step	Component	Files & Folders Created	Depends On	Purpose
1	Project Initialization	`.env`, `requirements.txt`, `Dockerfile`, `docker-compose.yml`	None	Sets up the project environment, including environment variables, dependencies, and containerization. `.env` stores sensitive configuration, `requirements.txt` lists Python dependencies, `Dockerfile` defines the container image, and `docker-compose.yml` orchestrates services like databases and Redis.
2	Core Setup	`app/main.py`, `app/core/config.py`	`.env`	Initializes the FastAPI application and loads environment variables. `main.py` is the entry point for the app, and `config.py` centralizes configuration settings like database URLs and API keys.
3	Database Integration	`app/db/session.py`, `app/db/base.py`, `app/db/mongo.py`	`config.py`	Configures database connections. `session.py` manages SQLAlchemy sessions, `base.py` defines the base ORM model, and `mongo.py` sets up MongoDB integration using Motor for async operations.

Step	Component	Files & Folders Created	Depends On	Purpose
4	Models & Schemas	`app/models/user.py`, `app/schemas/user.py`	`db/base.py`	Defines the data structure. `models/user.py` contains SQLAlchemy ORM models for database tables, and `schemas/user.py` defines Pydantic schemas for request/response validation and serialization.
5	Services & Routers	`app/services/user service.py`, `app/routers/users.py`	`models`, `schemas`, `db/session.py`	Implements business logic and API endpoints. `user_service.py` handles user-related operations (e.g., CRUD), and `routers/users.py` defines FastAPI routes for user-related endpoints.
6	Security & Middleware	`app/core/security.py`, `app/core/middleware.py`	`config.py`, `main.py`	Adds security and global middleware. `security.py` implements JWT authentication and password hashing, while `middleware.py` handles CORS, logging, and other global request/response processing.
7	ML Integration	`app/ml/models.py`, `app/ml/huggingface_api.py`	`config.py`	Integrates machine learning capabilities. `models.py` loads pre-trained AI/ML models (e.g., TensorFlow, PyTorch), and `huggingface_api.py` connects to Hugging Face for NLP tasks like text generation or classification.
8	Background Tasks	`app/worker/celery app.py`, `app/worker/tasks.py`	`config.py`	Enables asynchronous task processing. `celery_app.py` configures Celery for task queuing, and `tasks.py` defines background tasks (e.g., sending emails, processing data) that run independently of the main application.

Step	Component	Files & Folders Created	Depends On	Purpose
9	Testing & Deployment	`tests/test_users.py`, `README.md`	All previous	Ensures reliability and provides documentation. `test_users.py` contains unit and integration tests for user-related functionality, and `README.md` provides project documentation, including setup instructions and usage guidelines.

How Components Interact

Request-Response Cycle

1. **Client (Browser, Mobile, API Call)**

   > → Sends HTTP request to the FastAPI server.

2. **Middleware (Logging, CORS, Authentication)**

   > → Intercepts the request, logs it, checks authentication.

3. **Router (e.g., routers/users.py)**

   > → Determines the endpoint and calls the appropriate service.

4. **Dependency Injection (e.g., Depends(get_db))**

   > → Provides a database session, auth token, or other dependencies.

5. **Service Layer (services/user_service.py)**

   > → Handles business logic and interacts with models.

6. **Database Layer (models/user.py)**

   > → Queries PostgreSQL/MySQL (SQLAlchemy) or MongoDB.

7. **Response (JSON, ML Prediction, or API Call)**

   > → Data is returned as a Pydantic schema response.

Background Tasks (Celery & Redis)

1. **Router (POST /generate-report)**

   > → Calls Celery asynchronously.

2. **Celery Worker (tasks.py)**

> → Fetches job from Redis and processes it.

3. **Database (session.py)**

> → Saves result in PostgreSQL/MongoDB.

4. **Notification Sent**

> → API sends response once the task is completed.

Best Practices for Scalability & Production

1. **Database Best Practices**
    - Use PostgreSQL/MySQL with SQLAlchemy for structured data.
    - Use MongoDB for unstructured, flexible schema data.
    - Use connection pooling with SQLAlchemy.
2. **Dependency Injection Best Practices**
    - Use Depends(get_db) to manage DB sessions per request.
    - Use Depends(get_current_user) for authentication checks.
3. **Middleware Best Practices**
    - Enable CORS in core/middleware.py.
    - Add request logging in main.py.
4. **Asynchronous Processing**
    - Use Celery & Redis for long-running tasks.
    - Offload ML model inference to background tasks.
5. **Containerization & Deployment**
    - Use Docker for isolated environments.
    - Use Gunicorn with Uvicorn workers for performance.
    - Deploy with Kubernetes (K8s) or AWS ECS.
