FastAPI Backend Architecture Documentation

1. **Introduction**

This document provides an overview and detailed structure for organizing a FastAPI backend project. FastAPI is a modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python-type hints. It is designed to be flexible, efficient, and developer-friendly. This document outlines how to structure your FastAPI backend application for maintainability, scalability, and separation of concerns.

2. **Folder and File Structure**

A well-organized project structure helps in making the application scalable, maintainable, and easy to test. Below is a recommended structure for a FastAPI project:

```
fastapi-backend/
├── api/
|   ├── aws/                # AWS-related files, e.g., S3 integration
|   ├── core/               # Core functionality (database, settings, middlewares)
|   ├── migrations/         # Database migration scripts (e.g., Alembic)
|   ├── models/             # Database models (SQL Alchemy, Pydantic, etc.)
|   ├── routers/            # API route handlers/endpoints
|   ├── schemas/            # Pydantic schemas (for validation and serialization)
|   ├── utils/              # Utility functions/helpers
|   ├── __init__.py         # Package initializer
|   ├── logging_config.py   # Logging configuration file
|   └── main.py             # Entry point of the FastAPI app
├── tests/                  # Unit and integration tests
|   └── test_<feature>.py   # Individual test files
├── .gitignore              # Ignored files for git (e.g., virtualenv, secrets)
├── alembic.ini             # Alembic migration configuration
├── build.prop              # Build-related configuration
├── default.prop            # Default properties and settings
├── dev.env                 # Environment variables for development
├── Docker file             # Docker configuration for containerization
├── local.env               # Environment variables for local setup
├── makefile                # Script to automate common tasks (e.g., testing, linting)
```

```
├── README.md                # Project documentation
├── requirements.txt         # List of Python dependencies
├── server.log               # Server logs
├── setup.py                 # Script for packaging and distribution ()
├── version.py               # API version tracking
└── win_local.env            # Environment variables for Windows setup
```

3. **Key Components**

3.1 api/ Directory

The api/ directory contains all the core functionality of the FastAPI application, including models, routes, schemas, and utility functions.

- aws/: Contains AWS-related configurations and helper functions such as S3 storage integrations, SES (Simple Email Service), etc.
- core/: Holds core components such as the app configuration, database connection, middlewares, and dependencies.
    - Example Files:
        - database.py: Database connection settings (e.g., using SQL Alchemy).
        - config.py: Global application settings (could use Pydantic settings).
        - security.py: Authentication logic, JWT token management, etc.
- migrations/: Stores the database migration files (commonly managed with Alembic). You can generate new migration files when your database schema changes.
- models/: Contains SQL Alchemy models (or any other ORM models) that define the structure of your database tables.
    - Example: user.py, product.py for defining your User and Product tables.
- routers/: Defines all the API routes in separate files. Each router file represents a different resource (e.g., user, product, etc.).

- Example Files:
    - user_router.py: Defines all routes related to users (e.g., registration, login).
    - product_router.py: Defines all routes related to products (e.g., listing, adding products).

- schemas/: Contains Pydantic schemas used for request validation and response serialization. These schemas allow you to define data models with strict type validation.
    - Example Files:
        - user_schema.py: Defines user request/response validation models.
        - product_schema.py: Defines product validation models.
- utils/: Contains utility functions that are used across the application (e.g., hashing passwords, sending emails, generating tokens).
    - Example Files:
        - security.py: Utility functions for password hashing or token generation.
        - email.py: Utility for sending emails.
- logging_config.py: Configures how logging is handled in the application. This can be used to manage log levels, file outputs, or format specifications.
- main.py: The entry point of the FastAPI application where the app instance is created, routers are registered, and the app is run.

3.2 tests/ Directory

The tests/ directory contains unit and integration tests for the application. The tests are typically organized to mirror the structure of the api/ directory to ensure every component is covered.

- Example Files:
    - test_user.py: Tests for user-related functionality (registration, login, etc.).

3.3 Root-Level Files

- .gitignore: Specifies files and directories that should be ignored by version control (e.g., virtual environments, compiled files, or sensitive configuration files).
- alembic.ini: Configuration file for Alembic, the database migration tool. This is crucial for managing versioned changes to the database schema.
- Docker file: Describes how to build a Docker container for the FastAPI application. Docker ensures the app can run in a consistent environment across different platforms.
- requirements.txt: Lists all dependencies that the FastAPI application requires. These are installed via pip.
- make file: A script to automate tasks such as running tests, linting, formatting code, or building Docker images.
- README.md: Project documentation. This should include instructions for setting up and running the project, an overview of the API, and other important project information.
- .env Files (dev.env, local.env, win_local.env): These files store environment-specific configuration variables, such as database URLs, secret keys, or API keys. They are loaded into the application at runtime.

3.4 Setup File

The setup.py file is responsible for managing the packaging and distribution of the FastAPI project. It also includes optional Cython optimization to compile Python files to C extensions for improved performance. Below is a detailed explanation of the setup.py file.

Key Sections of setup.py:

- **Imports**:
    - setup tools: The standard library for packaging Python projects, which helps manage dependencies and package distribution.
    - cythonize: If available, Cython is used to compile the Python files into C extensions for performance optimization.

- **Extensions**: The Extension object defines the module or package to be compiled. It specifies the files and directories to be compiled and includes additional compile arguments like -O3 for optimization and -Wall for showing compiler warnings.

  Example :
  Extension("api.", ["api/**/*.py"], extra_compile_args=["-O3", "-Wall"])*
- **Cython Handling**: The script attempts to import cythonize from Cython. If Cython is available, it compiles the Python files; otherwise, it proceeds without compilation.
  ext_modules=cythonize(extensions) if cythonize else extensions
- **Install Requires**: The install_requires section lists all necessary dependencies for the FastAPI application. Common dependencies include FastAPI, Uvicorn, SQL Alchemy, and Alembic for database migrations.

Purpose of Key Sections:

- **Extension**: The Extension object allows you to define modules or packages that should be compiled using Cython. In this case, all Python files in the api/ folder are targeted. Compilation with Cython can boost performance by converting Python code into C extensions.
- **Cython Optimization**: By optionally using Cython, the setup file ensures that the Python code in api/ can be optimized for performance if Cython is installed. If Cython is not available, the project will still work normally, but without the performance improvements.
- **Dependencies**: The install requires list specifies external libraries that the project depends on, ensuring they are installed when the package is installed.

Running the Setup Script:

1. **Installing Dependencies**: To install all required dependencies for the FastAPI project:

   pip install -r requirements.txt

2. **Building the Project with Cython** (if installed): If Cython is installed and you wish to build the project with optimized extensions, run:

   python setup.py build_ext –inplace

3. **Installing the Package**: To install the FastAPI application as a package:

   pip install .

Benefits of Using setup.py with Cython:

- **Performance Optimization**: Cython can significantly improve the performance of certain parts of your FastAPI application, especially if they involve complex computations.
- **Packaging and Distribution**: With setup.py, the project can be easily packaged and distributed, making it easier to deploy or share the FastAPI app across environments or teams.
- **Flexibility**: The setup script is flexible, as it gracefully handles cases where Cython is not available, allowing the project to run as standard Python code.

4. **Environment Configuration**

Using .env files to manage configuration settings is essential for separating concerns between development, testing, and production environments. FastAPI applications can utilize python-dotenv to load environment variables from these .env files.

DATABASE_URL=postgresql://user:password@localhost/db_name

SECRET_KEY=your-secret-key

This ensures sensitive data such as passwords and API keys are not hard-coded in the source files and can be easily changed based on the environment.

5. Docker Setup

The Docker file ensures that your application can run in a Docker container, which can help in maintaining a consistent environment across different systems.

Example Docker file:

Use official python runtime as a parent image

FROM python:3.9

Set the working directory in the container

WORKDIR /app

Copy the current directory contents into the container at /app

```
COPY . /app

# Install the dependencies

RUN pip install -r requirements.txt

# Expose the port the app runs on

EXPOSE 8000

# Run the FastAPI application

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

6. Running the Application

Once the project is set up, you can run the FastAPI application in multiple ways.

1. **Using Uvicorn (Directly)**:

   uvicorn api.main:app --reload

   The --reload flag is used for development purposes, which auto-reloads the server on file changes.

2. **With Docker**:

   docker build -t fastapi-app

   docker run -d -p 8000:8000 fastapi-app

7. Conclusion

This FastAPI backend architecture provides a well-structured, modular, and scalable way to build modern APIs. By organizing the application into distinct modules, separating concerns, and using industry-standard tools like Docker, Alembic, and .env files, this setup ensures that the FastAPI project is maintainable and easy to scale for future growth.

Gokul C