**AI-Powered PowerPoint Generator: Project Folder Setup**

**Project Overview**

This backend project allows users to generate PowerPoint presentations automatically using AI-driven content generation tools like OpenAI's GPT-3/4. It follows two architectural patterns:

1. Python Microservice Architecture (using Flask)
2. C# Clean Architecture (Auth Service for authentication and authorization)

**Technology Stack (Python Service)**

* Backend Framework: Flask
* Database: PostgreSQL (SQLAlchemy ORM)
* Task Queue: Celery + Redis
* File Storage: AWS S3
* Authentication: JWT Tokens
* AI Content Generation: OpenAI API (GPT-3/4)
* Containerization: Docker
* Testing: Pytest
* Environment Management: Virtualenv

**Python Microservice Folder Structure**

/AIService

├── /app

│ ├── /api

│ │ ├── /v1

│ │ │ └── presentation\_api.py

│ │ └── /v2

│ │ └── presentation\_api\_v2.py

│ ├── /core

│ │ ├── ai\_model.py

│ │ └── content\_generator.py

│ ├── /services

│ │ ├── ai\_service.py

│ │ ├── file\_service.py

│ │ └── task\_queue\_service.py

│ ├── /infrastructure

│ │ ├── azure\_blob\_storage.py

│ │ ├── config.py

│ │ ├── logger.py

│ │ └── ppt\_formatter.py

│ ├── /models

│ │ ├── presentation\_model.py

│ │ └── task\_model.py

│ ├── /schemas

│ │ ├── presentation\_schema.py

│ │ └── task\_schema.py

│ └── /tests

│ ├── test\_ai\_service.py

│ ├── test\_api.py

│ ├── test\_file\_service.py

│ ├── test\_ppt\_formatter.py

│ └── test\_task\_queue.py

├── /docker

│ ├── Dockerfile

│ └── docker-compose.yml

├── /logs

├── /migrations

│ ├── env.py

│ └── /versions

├── .env

├── alembic.ini

├── README.md

├── requirements.txt

**C# Clean Architecture Folder Structure (AuthService)**

/AuthService

├── src

│ ├── AuthService.API

│ │ ├── Controllers

│ │ │ └── AuthController.cs

│ │ ├── Models

│ │ │ └── User.cs

│ │ └── Program.cs

│ ├── AuthService.Application

│ │ ├── Interfaces

│ │ │ └── IAuthService.cs

│ │ ├── Services

│ │ │ └── AuthService.cs

│ │ └── DTOs

│ │ └── AuthRequest.cs

│ ├── AuthService.Core

│ │ └── Entities

│ │ └── User.cs

│ ├── AuthService.Infrastructure

│ │ ├── Data

│ │ │ └── AuthRepository.cs

│ │ ├── JWT

│ │ │ └── JwtHelper.cs

│ │ └── Context

│ │ └── ApplicationDbContext.cs

│ └── AuthService.Tests

│ ├── AuthServiceTests.cs

│ └── AuthServiceMockTests.cs

├── docker

│ ├── Dockerfile

│ └── docker-compose.yml

├── README.md

└── requirements.txt

**Folder Structure Breakdown**

**Python (AIService)**

* app/api/v1: API version 1 for presentation generation
* app/api/v2: API version 2 with extended endpoints
* app/core: Core logic for AI model interaction and content generation
* app/services: Modular business logic including AI, file, and task services
* app/infrastructure: System integrations like storage, logging, configuration, and PowerPoint formatting
* app/models: SQLAlchemy database models
* app/schemas: Input/output validation using Marshmallow or similar libraries
* app/tests: Unit tests for all service and API components
* docker: Docker configuration for development and deployment
* logs: Runtime logs storage
* migrations: Alembic-based migration scripts and configuration
* .env: Environment variables used across the application
* alembic.ini: Configuration for Alembic migrations
* requirements.txt: Python dependencies

**C# (AuthService)**

* AuthService.API: ASP.NET Core Web API with controller logic and startup configuration
* AuthService.Application: Application layer handling interfaces, DTOs, and business logic
* AuthService.Core: Domain layer containing entity definitions
* AuthService.Infrastructure: Infrastructure concerns such as data access, JWT token utilities, and database context
* AuthService.Tests: Unit and mock testing for service and data logic
* docker: Containerization configuration for the AuthService
* README.md: Documentation for setup and usage
* requirements.txt: Optional, if Python interop is used

**Setup & Configuration (Python)**

**Step 1: Create Folder Structure**

Use terminal or script to generate the folders and files shown above.

**Step 2: Install Dependencies**

pip install -r requirements.txt

**Step 3: Set Up Database**

Use Alembic or Flask-Migrate to manage schema changes.

**Step 4: Run Celery**

celery -A app.tasks.celery worker --loglevel=info

**Step 5: Run the Application**

python app.py

**Setup & Configuration (C# AuthService)**

**Step 1: Run PowerShell Script**

This script generates folders, projects, and sets up Clean Architecture with references.

Set-ExecutionPolicy Bypass -Scope Process -Force

.\setup\_clean\_architecture\_authservice.ps1

**Step 2: Build and Run the API**

dotnet build

dotnet run --project src/AuthService.API

**Conclusion**

This hybrid backend architecture combines a Python microservice approach with a modular C# Clean Architecture AuthService. Python handles AI-based PowerPoint generation workflows, while C# ensures scalable and testable authentication workflows. This separation allows each service to scale independently, remain maintainable, and follow industry best practices.

Let us know if you would like to include CI/CD pipelines, architecture diagrams, or a deployment-ready Docker/Cloud setup.