**AI-Powered PowerPoint Generator: Backend Project Documentation**

**Project Overview**

This backend project provides an API service for generating AI-based PowerPoint presentations from user-provided prompts. The backend is built with **Flask** and uses **OpenAI** for content generation, **Celery** for background task processing, and **Blob-Storage** for file storage. The architecture is designed to be modular, scalable, and production-ready.

**1. Project Objectives**

* **Generate PPTs using AI**: Automatically generate PowerPoint presentations based on user input (e.g., topics, titles, content).
* **Cloud Storage**: Store generated PowerPoint files on AWS S3 for easy access and download.
* **Background Processing**: Use Celery and Redis to handle long-running tasks (like PPT generation) asynchronously.
* **Secure and Scalable API**: Expose a secure RESTful API with proper user authentication (JWT tokens).
* **AI Integration**: Use Google Gemini models to create content for slides.
* **Testing and Monitoring**: Implement unit and integration tests, logging, and error tracking for production environments.

**2. Folder Structure Breakdown**

**/backend**

The root directory for the backend project, containing all application logic, tests, and configuration.

**/app: Main Application Logic**

This is the core directory that contains the business logic and main functionality of the backend.

* /api: Handles all the API routes and controllers.
  + ppt\_controller.py: Manages user requests for generating PowerPoint presentations. It interacts with the ppt\_service.py to process the request and return the appropriate response.
  + auth\_controller.py: Manages user authentication and registration using JWT tokens.
  + status\_controller.py: Provides an endpoint for checking the status of long-running tasks (e.g., PPT generation).
  + user\_controller.py: Provides user-specific operations such as user details retrieval and profile management.
* /models: Contains the database models that define the application's data structure.
  + user.py: Defines the user model, with fields such as email, password, and user profile information.
  + presentation.py: Stores metadata related to each generated PowerPoint presentation, including user reference, filename, generation status, and storage URL.
  + task.py: Stores metadata related to background tasks, such as status, result, and timestamps.
* /services: Contains the application's core business logic, external integrations, and helper functions.
  + ai\_service.py: Provides functionality for interacting with the OpenAI API to generate content for the slides.
  + ppt\_service.py: Handles the logic to format content into a PowerPoint presentation using the python-pptx library.
  + file\_service.py: Responsible for uploading and downloading files to/from AWS S3.
  + auth\_service.py: Manages user authentication tasks, such as generating JWT tokens, validating credentials, etc.
* /tasks: Contains the Celery background tasks, which are responsible for handling long-running tasks asynchronously.
  + generate\_ppt.py: A Celery task that performs the PowerPoint generation process in the background, offloading heavy computation and preventing server timeouts.
* /schemas: Contains the input validation and serialization logic, often used to validate the request body or query parameters.
  + ppt\_schema.py: Defines the schema for the PowerPoint generation request, ensuring the input is valid.
  + user\_schema.py: Defines the schema for user-related requests, ensuring input data is correct.
* /utils: Utility functions for the backend, including custom error handling and JWT utility functions.
  + jwt\_utils.py: Provides functions for creating and verifying JWT tokens.
  + error\_utils.py: Centralized error handling and response formatting functions.
  + s3\_utils.py: AWS S3-related helper functions to manage file uploads and downloads.
* config.py: Configuration file that reads environment variables for sensitive information like API keys, database credentials, etc.

**/migrations: Database Migrations**

This folder contains the migration scripts for evolving the database schema over time using Flask-Migrate or a similar tool. These scripts keep track of the changes made to the database schema and allow you to apply changes to the production database in a controlled manner.

**/tests: Unit and Integration Tests**

Contains tests for the application to ensure its correctness and functionality.

* /unit\_tests: Tests for individual components of the application, such as services and utility functions.
* /integration\_tests: Tests for ensuring the interaction between different parts of the application (e.g., API endpoints, database, external services).
* /fixtures: Contains mock data used in tests, such as user credentials or sample PowerPoint content.
* pytest.ini: Configuration file for pytest, which defines test setup, environment variables, and other settings.

**/docker: Docker Configuration**

* Dockerfile: The Dockerfile to containerize the backend application, specifying the image to use, dependencies to install, and how to run the application.
* docker-compose.yml: Defines the multi-container environment needed for the project, including services like the Flask app, Redis (for Celery), and PostgreSQL.

**/scripts: Automation and Deployment**

* deploy.sh: A deployment script that automates tasks like pushing the application to a server, running migrations, and configuring environment variables.

**Other Key Files**

* requirements.txt: Lists the Python dependencies required to run the backend application.
* .gitignore: Specifies which files and directories should be ignored by version control.
* README.md: The documentation for developers, explaining the project setup, installation, and usage instructions.
* .env: A configuration file that holds environment-specific variables like API keys, database URIs, and secret keys.

**3. Functional Flow**

**User Flow**

1. **User Registration and Authentication**:
   * A new user registers or logs in via API endpoints (POST /register, POST /login).
   * Upon successful login, a JWT token is generated and returned to the user.
2. **PowerPoint Generation**:
   * The user submits a request to generate a PowerPoint by sending a POST /generate request, passing in their desired topic, content, or any specific instructions.
   * The backend validates the request data and triggers a Celery task (generate\_ppt.py), which performs the actual generation of the PowerPoint presentation using OpenAI and python-pptx.
   * The presentation is uploaded to AWS S3, and the URL is stored in the database.
3. **Checking Task Status**:
   * The user can check the status of their PowerPoint generation by sending a GET /status/{task\_id} request. The backend queries the task status from the database or Celery.
4. **Downloading PowerPoint**:
   * Once the task is completed, the user can download the generated PowerPoint presentation using the URL stored in the database.

**Background Processing with Celery**

* **Celery** is used for handling the PowerPoint generation asynchronously. When a user submits a request for PPT generation, the task is added to the Celery queue, which allows the Flask app to immediately respond without waiting for the AI model to generate content and create the PPT.

**4. Security and Authentication**

* **JWT Authentication**: All API requests (except registration/login) require the user to provide a valid JWT token in the authorization header. The token is validated before any operation is performed.
* **Environment Variables**: Sensitive keys like API keys, database URIs, and secret keys are stored in .env files and not hardcoded into the codebase to enhance security.

**5. Deployment Instructions**

**Setup**

1. Clone the repository and navigate into the backend folder.
2. Create a virtual environment and activate it:

bash

Copy

python -m venv venv .\venv\Scripts\activate *# For Windows* source venv/bin/activate *# For Linux/Mac*

1. Install the dependencies:

bash

Copy

pip install -r requirements.txt

**Environment Variables**

Create a .env file with the following required variables:

* FLASK\_APP
* FLASK\_ENV
* SECRET\_KEY
* JWT\_SECRET\_KEY
* OPENAI\_API\_KEY
* AWS\_ACCESS\_KEY\_ID
* AWS\_SECRET\_ACCESS\_KEY
* DATABASE\_URL

**Running the Application**

1. Start Redis (for Celery):

bash

Copy

redis-server

1. Start Celery Worker:

bash

Copy

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

1. Run the Flask application:

bash

Copy

python app.py

**6. Conclusion**

This documentation outlines the setup, folder structure, and key components of the **AI-Powered PowerPoint Generator** backend project. The design emphasizes scalability, modularity, and ease of maintenance by following industry-level best practices.

Feel free to modify, extend, and customize the architecture as per your requirements. This project can serve as a robust foundation for building sophisticated AI-powered applications in various domains.

Let me know if you need further clarification or more information!