Weekly Report: Oudarja Barman Tanmoy Alpha AI

Week-06 (April 28 - May 01)

Digital Resource Management System Development : After completion of a session of AWS I have understood the fundamentals of cloud computing and why cloud platforms like AWS are widely adopted in scalable applications. Then a project titled "DRMS" was assigned and to proceed with the assigned project following the given requirements, I explored key AWS services (DynamoDB and s3 storage) relevant to the DRMS project.

A. DynamoDB:

1. Learned how to use AWS DynamoDB for storing structured, serverless NoSQL data.
2. Implemented CRUD operations (Create, Read, Update, Delete) using Python with the boto3 SDK.
3. Built a basic CRUD application using Python and DynamoDB as a personal practice to strengthen backend development skills.
4. Structure data with partition keys and nested attributes. This hands-on exercise served as preparation for the DRMS backend integration.

B. S3 storage :

1. Learned to interact with **AWS S3** for storing employee-related images.
2. Used **pre-signed URLs** for secure and temporary access to images.
3. Ensured proper connection between uploaded image URLs and metadata stored in DynamoDB.

Learned FastAPI :

a) Core FastAPI Concepts :

1. Understood the basic structure of a FastAPI application (including app = FastAPI()).
2. Learned to create routes (GET, POST, PUT, DELETE) to handle different HTTP methods.

b) Path & Query Parameters :

1. Practiced how to define dynamic path parameters (e.g., /employee/{id}).
2. Used query parameters to filter or modify requests (e.g., /employee?name=Alice).

c) Request & Response Models (Pydantic) :

1. Learned to define and use Pydantic models for:
2. Validating incoming request data
3. Structuring and validating response output

d) CRUD Operations : Built and tested FastAPI endpoints to:

1. Create new entries (`POST`)
2. Read all or filtered data (`GET`)
3. Update existing records (`PUT`)
4. Delete records (`DELETE`)

Planned project architecture (MVC pattern) :

DRMS project follows a loosely coupled MVC (Model-View-Controller) style architecture. So before starting , a MVC like pattern was thought of and how to implement following this pattern , what file should be in what type , and these decisions were made. I have planned a **loosely coupled MVC (Model-View-Controller)** style architecture to ensure separation of concerns. Established clear structural guidelines:

- **Models**: Data schemas using Pydantic for both request/response and internal logic.

- **Views**: API routes acting as entry points for HTTP requests.

- **Controllers/Services**: Core business logic like DB interaction, image handling, etc.

File and folder structures were designed for **scalability and maintainability**.