Assignment 3 – Question 3

Job Portal RESTful API

Objective

The goal of this project was to design and build a simple **RESTful API** for a job portal where **employers** can post jobs and **job seekers** can apply and upload resumes. The API follows REST principles using clear endpoints, HTTP methods, and stateless interactions.

Overview

The project, called **Job Portal API**, was developed with **FastAPI**. It automatically provides an interactive **Swagger UI** for testing endpoints. The application supports creating and viewing jobs, submitting applications, and uploading resumes which can be accessed through direct links.

Project Directory Structure:

```
job-portal-api/
│
├── app/
│   ├── main.py                 # Entry point
│   ├── routes/                 # RESTful endpoints
│   │   ├── jobs.py
│   │   ├── employers.py
│   │   ├── seekers.py
│   │   └── applications.py
│   ├── models.py               # Database models
│   ├── database.py             # DB connection setup
│   ├── storage/                # Resume uploads folder
│   └── requirements.txt
│
├── Dockerfile
├── docker-compose.yml
└── docs/                       # Documentation and setup guides
```

RESTful API Design

Core Resources:

- `/jobs` → Job postings
- `/employers` → Employer accounts and job management
- `/seekers` → Job seeker profiles
- `/applications` → Job applications linking seekers and jobs
- `/upload-resume/` → File upload endpoint

Example Endpoints:

Functionality	Method	Endpoint	Description
List all jobs	`GET`	`/jobs`	Retrieve all job listings
Create job	`POST`	`/employers/{id}/jobs`	Employer posts new job
Apply for job	`POST`	`/jobs/{id}/applications`	Job seeker applies
Upload resume	`POST`	`/upload-resume/`	Uploads resume (PDF)
Access resume	`GET`	`/storage/{filename}`	Retrieve stored resume

How It Works

1. Run `docker-compose up --build` to start the service.
2. Open http://localhost:8000/docs to use Swagger.
3. Test the API by uploading resumes or posting jobs.

Resumes are stored in the local **`app/storage`** folder and remain even after restarting the container.

Technologies Used

- **FastAPI** for API framework
- **Docker Compose** for deployment
- **SQLite** for storage
- **Python 3.10**

Conclusion

The Job Portal API demonstrates a clean RESTful design with well-structured endpoints, Swagger documentation, and Docker deployment. It provides a simple local solution for managing job posts and applications, with the potential for cloud deployment and authentication in future versions.

Question 4

Evaluation of AviationStack REST API

Overview

The **AviationStack API** is a public REST service that provides global aviation data, including real-time and historical flight information. It offers multiple endpoints under a base URL such as `https://api.aviationstack.com/v1` and uses API keys for access.

Major API Endpoints

- **/flights** – Real-time and historical flight data
- **/routes** – Airline route details
- **/airports** – Airport information (IATA, ICAO, coordinates)
- **/airlines** – Airline details
- **/airplanes** – Information about individual aircraft
- **/aircraft_types** – Aircraft model and type information
- **/taxes** – Aviation tax data
- **/cities** – City details (IATA city codes)
- **/countries** – Country information (ISO codes, population, etc.)
- **/timetable** – Daily arrival/departure schedules for airports
- **/flightsfuture** – Future flight schedules beyond the current week

(a) Resource Naming

The API follows RESTful naming conventions using **plural nouns** for resources, such as `/flights`, `/airlines`, and `/airports`. Each resource represents a collection and can be filtered through query parameters like `dep_iata` or `airline_iata`. The structure is simple and versioned (`/v1/`), ensuring clarity and backward compatibility. However, the endpoint `/flightsfuture` slightly breaks the naming consistency, which could have been `/flights/future` for better hierarchy.

Verdict: Mostly RESTful naming; minor inconsistency with `/flightsfuture`.

(b) HTTP Methods

All endpoints use the **GET** method, which aligns perfectly with the API's data-retrieval purpose. The API is read-only and does not support POST, PUT, or DELETE requests. Each call is idempotent and safe, using proper status codes like 200 (OK) or 401 (Unauthorized) for errors when the access key is missing.

Verdict: Correct and consistent use of HTTP methods.

(c) Statelessness

Every request requires an **access_key** parameter for authentication and includes all necessary information. The server stores no client context, and pagination is handled via parameters like `limit` and `offset`. This design ensures that each request is independent, adhering to REST's stateless principle.

Verdict: Fully stateless; no server-side session or state retention.

(d) HATEOAS Implementation

The API does **not** include hyperlinks or embedded links in its responses. Clients must manually build URLs for related data using identifiers (e.g., an airport IATA code). While responses include pagination metadata, there are no `self`, `next`, or `prev` links.

Verdict: HATEOAS is not implemented.

Summary Table

REST Principle	Evaluation	Comment
Resource Naming		Clear and consistent, except `/flightsfuture`
HTTP Methods		Correct GET usage, proper status codes
Statelessness		Each request carries full context
HATEOAS		No hypermedia links or navigation

Conclusion

AviationStack's API is a well-structured RESTful service with properly named resources, consistent GET-based access, and stateless communication. The only missing aspect is HATEOAS support, which would enhance discoverability. Overall, it effectively demonstrates key REST principles and provides reliable aviation data for developers.