TI-IS Technical Test

Introduction

The following test simulates a real IT-IS project

Resource type	Description
Service Rest API	https://dummyjson.com/users
Programming language	Your preference (python, apache camels, golang is a plus)
IDE	Your preference
Repository	Public into github or gitlab
Database	sqlite
Database UI	https://sqlitebrowser.org/
Frontend technology UI	Your preference

Assumptions:

- Solution should be ready to be deployed on a linux environment, preferably running a dockerized environment using a alpine debian image

Backend

Important: Configuration parameters required as an external file (.env file), consider for example: token as a configuration parameter, credential database, SFTP credentials or keys, and other parameters. Include a README file showing artifact details, including component diagram, programming practices put in place

Part I

TWO of the below is mandatory, bonus points will be added, if any other or all are also completed.

- Implement a SFTP using complexity authentication methods, such as: SSH Key Authentication, SSH Key Authentication + passphrase, etc. Choose one of these options, no user and password auth is allow
- Data encryption on the file that will be pushed to the SFTP: obfuscate data, symmetric encryption, asymmetric encryption, etc.
- Implement decoupled components to save files on the SFTP or to save on database, using message queue technologies, such as: Amazon MQ, Google Pub/sub, Redis Pub/Sub, etc.

Phase 1: The Resilient Extractor

This component is responsible for reliably fetching a complete dataset of users from the API.

Requirements:

1. **Scheduled Execution:** The process must be configured to run automatically once a day.
2. **Resumable Fetching:** The API uses `limit` and `skip` for pagination. Your extractor must be designed to be resumable. If the job fails midway through fetching pages, it should be able to restart from the last successfully processed batch (i.e., the last `skip` value) instead of starting over from the beginning.
    - Persist the number of records successfully fetched (e.g., in a state file).
    - On each run, use this number as the initial `skip` value to resume or start fresh.
    - Upon successful completion of the entire extraction, reset the state for the next day's run.
3. **Full Data Extraction via Pagination:** The API returns a `total`, `skip`, and `limit` in its response. Your extractor must intelligently handle pagination by making sequential requests (e.g., `?limit=100&skip=0`, `?limit=100&skip=100`, etc.) until all `total` users have been fetched.
4. **Error Handling & Retries:**
    - If the API returns a non-200 status code, the extractor should retry the request up to **3 times** with an exponential backoff strategy (e.g., wait 5s, then 15s, then 45s).
    - If all retries for a specific page fail, the process should exit with a clear error message and a non-zero exit code, ensuring the `skip` state is **not** advanced.
5. **Output:** Write the raw, fetched user records to a file named `raw_users/records_[YYYYMMDD_HHMMSS].jsonl` (JSON Lines format, where each line is a separate JSON user object).

Phase 2: Transformation and Validation

This component processes the raw user data, validates it against business rules, and prepares it for loading. It should be an independent service/script triggered by the presence of a new file in the `raw_users` directory.

Requirements:

1. **Schema Validation:** For each JSON record from the `.jsonl` file, validate it against a predefined schema. A valid user record must contain:
    - `id` (integer)
    - `firstName` (string, non-empty)
    - `email` (string, must be a valid email format)
    - `age` (number, between 18 and 65 inclusive)
    - `company.department` (string, non-empty)
2. **Data Enrichment:** The user records contain a `company.department` field (e.g., "Marketing", "Engineering"). Enrich this data by mapping the department name to a

`department_code` using a provided `departments.csv` lookup file (you can create a sample of this file).

3. **Dead-Letter Queue for Invalid Data:**
    - If a record fails validation (e.g., age outside the valid range, missing email), move the invalid record to a separate file: `dlq/invalid_users_[YYYYMMDD_HHMMSS].jsonl`.
    - Each line in this DLQ file must include the original record plus a new field, `error_reason`, explaining why it failed.
4. **Output:** Write all valid, transformed, and enriched records to `processed_users/etl_[YYYYMMDD_HHMMSS].jsonl`.

Phase 3: Saving

1. Create a **second independent** deployment (using as preference: Java Apache Camel, Python or any integration framework or language), and listening the queue (**implement a in subscription to the queue**) process the messages on the queue:

- Save to database the produced files: `raw_users/records_[YYYYMMDD_HHMMSS].jsonl` and `dlq/invalid_users_[YYYYMMDD_HHMMSS].jsonl` and `processed_users/etl_[YYYYMMDD_HHMMSS].jsonl` adding columns with id and date insertion.
- Upload on a SFTP the generated files

Outcomes:

Upload project github or gitlab (share URL), including a docker-compose to install the deployment component: task service or task deployment, database, and extra decoupled components.

Bonus items:

- Include test script using Test Driven Development (TDD) framework, and include their reference on the deployment script to test components and services

Frontend

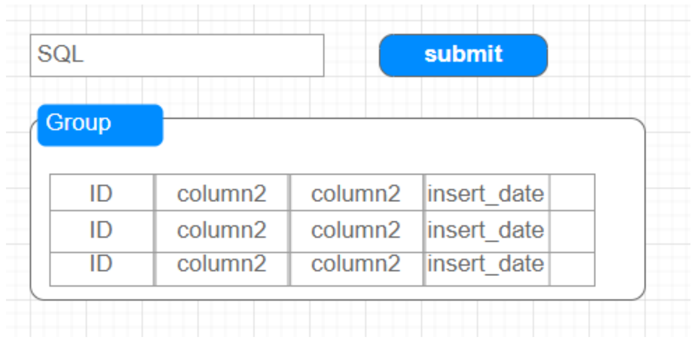
Part II

ONE of the below is mandatory, bonus points will be added, if any other or all are also completed.

- Create a login and signup pages to authenticate user and create credentials, including the feature to subscribe and generate OTP using Google Authenticator
- Authenticate using Google OAuth 2.0 for Client-side Web Applications

<u>Important</u>: Configuration parameters required as an external file (.env file), consider for example: configuration parameter is token, credential database, OTP or Google Authentication settings, backend base path url, etc. Include a README file showing artifact details, including component diagram, programming practices put in place

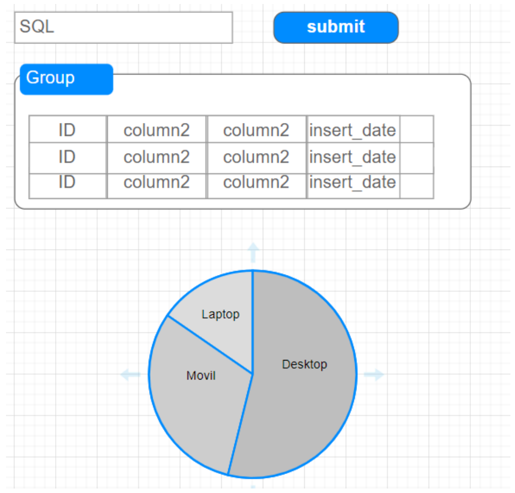
1. Create a web project where you display the saved information, allowing filter by process data using a drop list pointed to the header table; it is a must to build Back-End and Front-End components on separated github repositories.



ID	column2	column2	insert_date	
ID	column2	column2	insert_date	
ID	column2	column2	insert_date	

2. Add at least **two** charts showing the summary information (optional is a plus)

Noted: the graphs is your preference



ID	column2	column2	insert_date	
ID	column2	column2	insert_date	
ID	column2	column2	insert_date	

Outcomes:
Upload project github or gitlab (share URL), including a docker-compose to install the deployment component: task service or task deployment, database, and extra decoupled components.

Bonus items:

- Include test script using Test Driven Development (TDD) framework, and include their reference on the deployment script to test components and services