## Import Modules

* **FastAPI**: Used for creating web API.
* **StaticFiles**: Serves static files like images.
* **json**: Handles JSON data.
* **random**: Simulates randomized health checks.
* **sqlite3**: Stores health check results persistently.
* **matplotlib.pyplot**: Creates the graphical representation of the DAG.
* **asyncio**: Handles asynchronous operations (health checks).
* **collections.deque**: Implements Breadth-First Search (BFS).
* **os**: Manages file operations.
* **networkx**: Used to generate the DAG and draw it.

## Define the Directed Acyclic Graph (DAG)

* Define a sample DAG structure in JSON format.
* Each key is a **node**, and the corresponding value is a **list of dependent nodes**. Here, each node is a multi-level correlated component.

## Initialize the Database

* Connect to (or create) an SQLite database named health\_check.db.
* Create a table health\_status with:

1. node (Primary Key) → The node name.
2. status → Either "healthy" or "unhealthy".

* Ensure the table exists before running the health check.

## Simulate Health Checks

Simulate a health check by:

1. Introducing a random delay (0.5 to 1.5 sec).
2. Randomly returning "healthy" or "unhealthy".

## Store Health Check Results

* Save results to the SQLite database.
* **REPLACE INTO** ensures:

1. If a node exists, it updates its status.
2. If a node doesn't exist, it inserts a new record.

## Process the DAG using BFS

* Use **Breadth-First Search (BFS)** to traverse the graph.
* **Asynchronously** perform health checks on each node.
* Store and return results.

## Visualize the Graph

* Use **networkx** to draw the graph.
* Nodes are colored based on health:

1. Green → Healthy
2. Red → Unhealthy

* Save the image as **static/dag\_health.png**

## API Endpoint

* Retrieve the DAG.
* Run the health check.
* Generate and save the DAG visualization.
* Return:

1. The **health status** of each node.
2. A **URL** to the generated image.