# Stateless Claims Management System

## **1. Define Basic Entities**

* **Policyholder**: Represents an individual who owns an insurance policy.
  + Attributes:policyholder\_id, name, date\_of\_birth, address, phone
* **Policy**: Represents an insurance agreement between the policyholder and the insurer.
  + Attributes: policy\_id, policyholder\_id, start\_date, end\_date, premium
* **Claim**: Represents a request for payment based on the terms of the insurance policy.
  + Attributes: claim\_id, policy\_id, date\_of\_claim, claim\_amount, status

## 2. Low-Level Design (LLD)

### a. Understanding

The Claims Management System (CMS) is designed to manage and process insurance claims effectively. This system will handle the creation, tracking, and updating of claims made by policyholders. Without the use of persistent storage, the system will rely on in-memory data structures to manage data during runtime, suitable for demonstration or small-scale applications.

### b. Implementation Approach

* **Data Structures Used**:
  + **Dictionaries**: To store policyholders, policies, and claims using unique IDs as keys.
  + **Classes**: Each entity such as Policyholder, Policy, and Claim will be represented as a class with relevant attributes.
* **System Operations**:
  + **Add, Update, and Delete operations** for Policyholders, Policies, and Claims.
  + **View Claims**: Filter and view claims based on status, date, policyholder, etc.

## 3. CRUD Functions

I've extended the `ClaimsManagementSystem` class with CRUD (Create, Read, Update, Delete) functions for each of the main entities: `Policyholder`, `Policy`, and `Claim`. These functions handle adding, retrieving, updating, and deleting records within our in-memory data structures.

## 4. Validation and Business Rules

* **General Validations**:
  + **Date Validation**: Ensures all date inputs are valid and correctly formatted. This is crucial for avoiding errors related to date operations.
  + **Required Fields**: Checks like ensuring all necessary information for a policyholder is present before adding them to the system.
* **Specific Business Rules**:
  + **Policy Existence**: Ensures that the policyholder exists before a policy can be created.
  + **Date Consistency**: Checks that the policy start date is not later than the end date.
  + **Claim Amount**: Ensures that the claim amount does not exceed the total policy premium.

## 5. API Development

### **Request Descriptions:**

#### **1. Home**

* **Endpoint**: GET /
* **Description**: Returns a simple greeting message. This endpoint can be used to check the availability and responsiveness of the API.

#### **2. Add Policyholder**

* **Endpoint**: POST /policyholders
* **Description**: Adds a new policyholder to the system. Requires a JSON payload containing the policyholder's details such as ID, name, date of birth, address, and phone number. Returns a success message upon successful addition.

#### **3. Get Policyholder**

* **Endpoint**: GET /policyholders
* **Description**: Retrieves details of a specific policyholder by their ID. The policyholder ID must be provided as a query parameter. Returns the policyholder's details if found.

#### **4. Update Policyholder**

* **Endpoint**: PUT /policyholders/<policyholder\_id>
* **Description**: Updates information for an existing policyholder identified by their policyholder ID. Accepts partial or full JSON data for fields like name, date of birth, address, and phone.

#### **5. Delete Policyholder**

* **Endpoint**: DELETE /policyholders/<policyholder\_id>
* **Description**: Deletes a policyholder from the system using their ID. Returns a confirmation message if the deletion is successful.

#### **6. Add Policy**

* **Endpoint**: POST /policies
* **Description**: Adds a new policy to the system. Requires a JSON payload with details such as policy ID, policyholder ID, start date, end date, and premium. Validates the existence of the policyholder and date formats before addition.

#### **7. Get Policy**

* **Endpoint**: GET /policies
* **Description**: Retrieves a specific policy by its ID. The policy ID must be provided as a query parameter. Returns the policy details if found.

#### **8. Update Policy**

* **Endpoint**: PUT /policies/<policy\_id>
* **Description**: Updates details of an existing policy. Supports modification of start date, end date, and premium amount through JSON payload.

#### **9. Delete Policy**

* **Endpoint**: DELETE /policies/<policy\_id>
* **Description**: Removes a policy from the system using its ID. Provides a success message upon successful deletion.

#### **10. Add Claim**

* **Endpoint**: POST /claims
* **Description**: Registers a new claim associated with a specific policy. Requires JSON payload including claim ID, policy ID, date of the claim, claim amount, and status. Validates policy existence and claim amount constraints.

#### **11. Get Claim**

* **Endpoint**: GET /claims
* **Description**: Fetches details of a specific claim by its ID. The claim ID should be provided as a query parameter. Returns the claim's details if available.

#### **12. Update Claim**

* **Endpoint**: PUT /claims/<claim\_id>
* **Description**: Updates an existing claim's details such as date of claim, amount, or status. Accepts JSON payload with updated information.

#### **13. Delete Claim**

* **Endpoint**: DELETE /claims/<claim\_id>
* **Description**: Deletes a claim from the system using its ID. Confirms the deletion with a success message.