Chidambaram S **DATE: 26/10/2024**

Chidambaram S - Assignment Submission for Zeotap Internship

PROBLEM STATEMENT - 1

Application 1 : Rule Engine with AST

Objective:

Develop a simple 3-tier rule engine application(Simple UI, API and Backend, Data) to determine user eligibility based on attributes like age, department, income, spend etc.The system can useAbstract Syntax Tree (AST) to represent conditional rules and allow for dynamic creation,combination, and modification of these rules.

Data Structure:

- Define a data structure to represent the AST.
- The data structure should allow rule changes
- E,g One data structurecould be Node with following fields
    - type: String indicating the node type ("operator" for AND/OR, "operand" for conditions)
    - left: Reference to another Node(left child)
    - right: Reference to another Node(right child for operators)
    - value: Optional value for operand nodes (e.g., number for comparisons)

Data Storage

- Define the choice of database for storing the above rules and application metadata
- Define the schema with samples.

Sample Rules:

- rule1 = "((age > 30 AND department = 'Sales') OR (age < 25 ANDdepartment = 'Marketing')) AND (salary > 50000 OR experience >5)"
- rule2 = "((age > 30 AND department = 'Marketing')) AND (salary >20000 OR experience > 5)"

API Design:

1. **numcreate_rule(rule_string):** This function takes a string representing a rule (as shown in the examples) and returns a Nodeobject representing the corresponding AST.
2. **combine_rules(rules):** This function takes a list of rule strings and combines theminto a single AST. It should consider efficiency and minimize redundant checks. You canexplore different strategies (e.g., most frequent operator heuristic). The function should return the root node of the combined AST.

3. **evaluate_rule(JSON data):** This function takes a JSONrepresenting the combined rule's AST and a dictionary data containing attributes (e.g., data = {"age": 35, "department": "Sales", "salary": 60000, "experience": 3}). The function should evaluate the rule against the provided data and return True if the user isof that cohort based on the rule, False otherwise.

Test Cases:

1. Create individual rules from the examples using create_ruleand verify their AST representation.
2. Combine the example rules using combine_rulesand ensure the resulting AST reflects the combined logic.
3. Implement sample JSON data and test evaluate_rulefor different scenarios.
4. Explore combining additional rules and test the functionality.

Bonus:

- Implement error handling for invalid rule strings or data formats (e.g., missing operators, invalid comparisons).
- Implement validations for attributes to be part of a catalog.
- Allow for modification of existing rules using additional functionalities within create_ruleor separate functions.This could involve changing operators, operand values, or adding/removing sub-expressions within the AST.
- Consider extending the system to support user-defined functions within the rule language for advanced conditions (outside the scope of this exercise).

SOLUTION - 1

Rule Engine with Abstract Syntax Tree (AST)

Project Overview

Objective

The goal of this project is to develop a simple 3-tier rule engine application comprising a UI, API, and backend data management system. This engine determines user eligibility based on attributes such as age, department, income, and spending. The project utilizes an Abstract Syntax Tree (AST) to represent conditional rules and allows for dynamic creation, combination, and modification of these rules.

Technologies Used

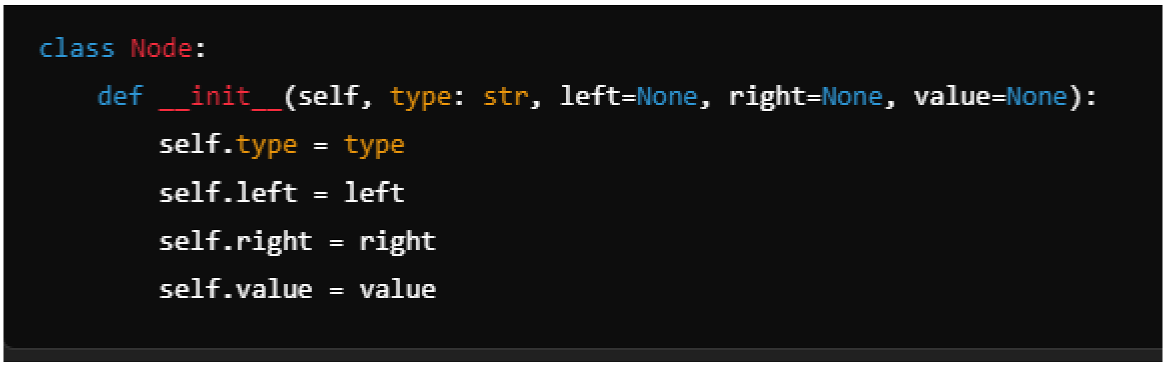
- **Framework**: Django
- **Database**: SQLite (or any relational database)
- **Frontend**: Simple HTML/CSS for UI
- **API**: RESTful API for rule management and evaluation

GitHub Repository
You can view the complete project codebase, build scripts, and configurations in my GitHub repository**: GitHub Repository Link**.

Core Concepts

Data Structure

The core data structure used to represent the AST is defined as follows:

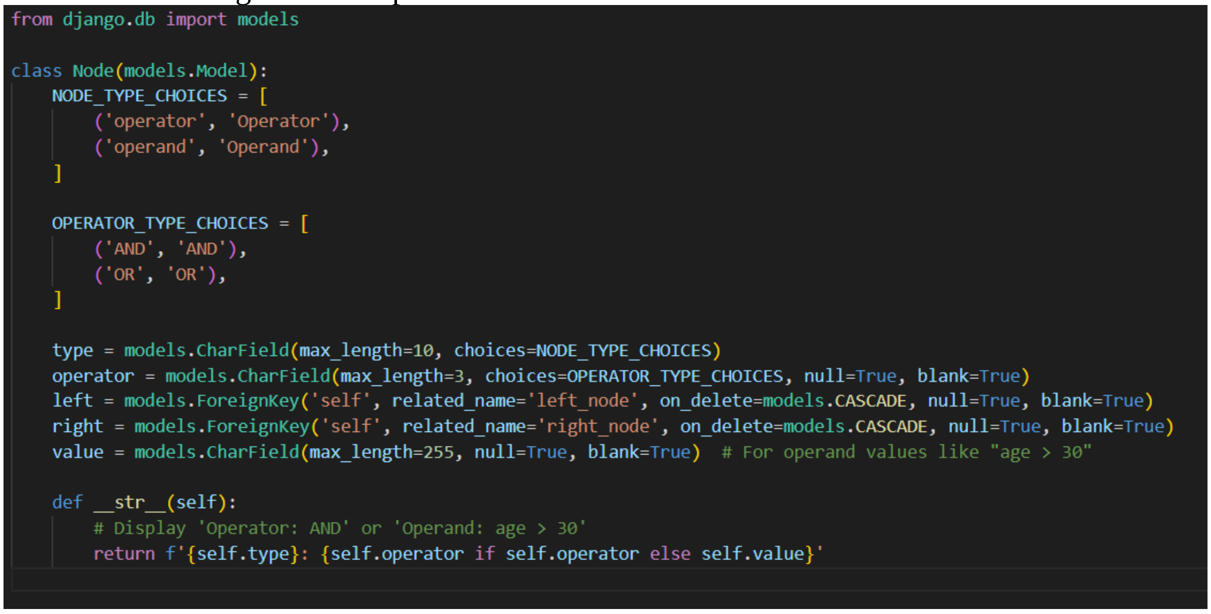


```
class Node:
    def __init__(self, type: str, left=None, right=None, value=None):
        self.type = type
        self.left = left
        self.right = right
        self.value = value
```

This structure allows for easy manipulation of rules, enabling changes and modifications to the AST dynamically.

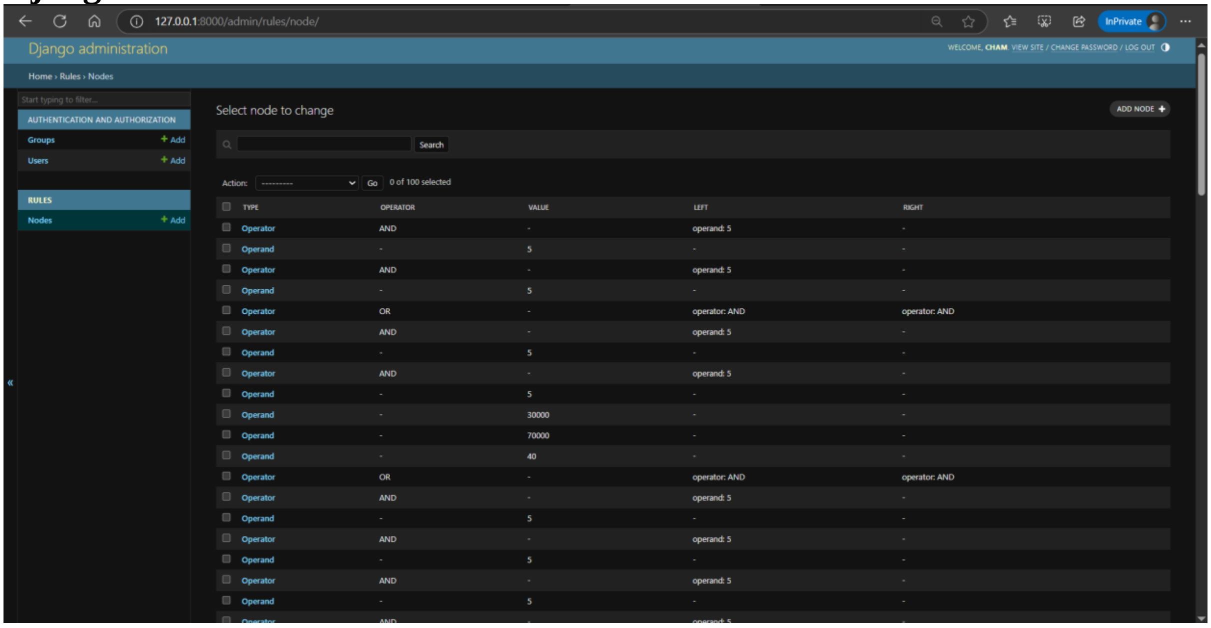
Data Storage

For storing the rules and application metadata, I have chosen SQLite as the database. Below is the schema design with sample rules:



```python
from django.db import models

class Node(models.Model):
    NODE_TYPE_CHOICES = [
        ('operator', 'Operator'),
        ('operand', 'Operand'),
    ]

    OPERATOR_TYPE_CHOICES = [
        ('AND', 'AND'),
        ('OR', 'OR'),
    ]

    type = models.CharField(max_length=10, choices=NODE_TYPE_CHOICES)
    operator = models.CharField(max_length=3, choices=OPERATOR_TYPE_CHOICES, null=True, blank=True)
    left = models.ForeignKey('self', related_name='left_node', on_delete=models.CASCADE, null=True, blank=True)
    right = models.ForeignKey('self', related_name='right_node', on_delete=models.CASCADE, null=True, blank=True)
    value = models.CharField(max_length=255, null=True, blank=True)  # For operand values like "age > 30"

    def __str__(self):
        # Display 'Operator: AND' or 'Operand: age > 30'
        return f'{self.type}: {self.operator if self.operator else self.value}'
```

Django Admin Panel Interface:



127.0.0.1:8000/admin/rules/node/

Django administration

WELCOME, CHAM. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Rules › Nodes

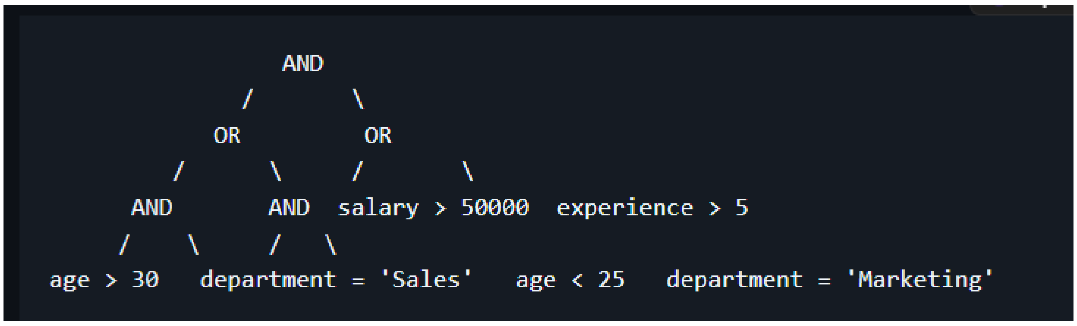
Select node to change

Action: --------- Go 0 of 100 selected

TYPE	OPERATOR	VALUE	LEFT	RIGHT
Operator	AND	-	operand: 5	-
Operand	-	5	-	-
Operator	AND	-	operand: 5	-
Operand	-	5	-	-
Operator	OR	-	operator: AND	operator: AND
Operator	AND	-	operand: 5	-
Operand	-	5	-	-
Operator	AND	-	operand: 5	-
Operand	-	5	-	-
Operand	-	30000	-	-
Operand	-	70000	-	-
Operand	-	40	-	-
Operator	OR	-	operator: AND	operator: AND
Operator	AND	-	operand: 5	-
Operand	-	5	-	-
Operator	AND	-	operand: 5	-
Operand	-	5	-	-
Operator	AND	-	operand: 5	-
Operand	-	5	-	-
Operator	AND	-	operand: 5	-

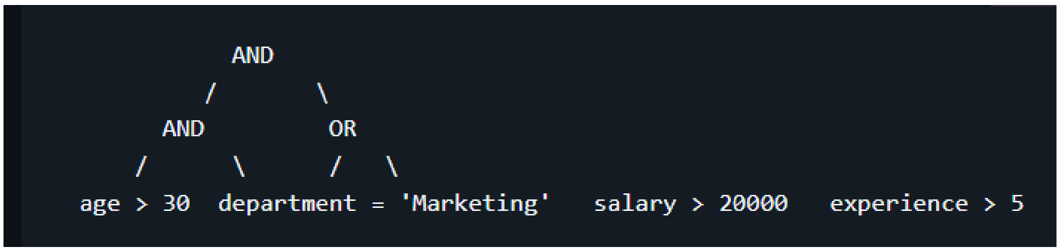
Sample Rules

- **Rule 1**: ((age > 30 AND department = 'Sales') OR (age < 25 AND department = 'Marketing')) AND (salary > 50000 OR experience > 5)



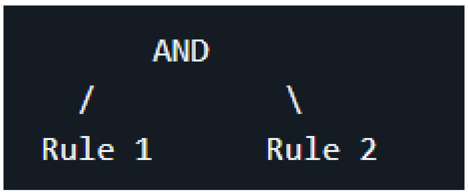
```
                  AND
                /     \
              OR        OR
           /     \    /      \
        AND      AND  salary > 50000  experience > 5
       /   \     /  \
  age > 30  department = 'Sales'   age < 25   department = 'Marketing'
```

- **Rule 2**: ((age > 30 AND department = 'Marketing')) AND (salary > 20000 OR experience > 5)



```
                AND
              /     \
           AND        OR
         /     \     /  \
   age > 30  department = 'Marketing'   salary > 20000   experience > 5
```

  - **Combined Rules**: Assuming both rules (Rule 1 and Rule 2) are combined with an AND operator.



```
      AND
   /        \
Rule 1      Rule 2
```

API Design

The following API functions are implemented in the project:

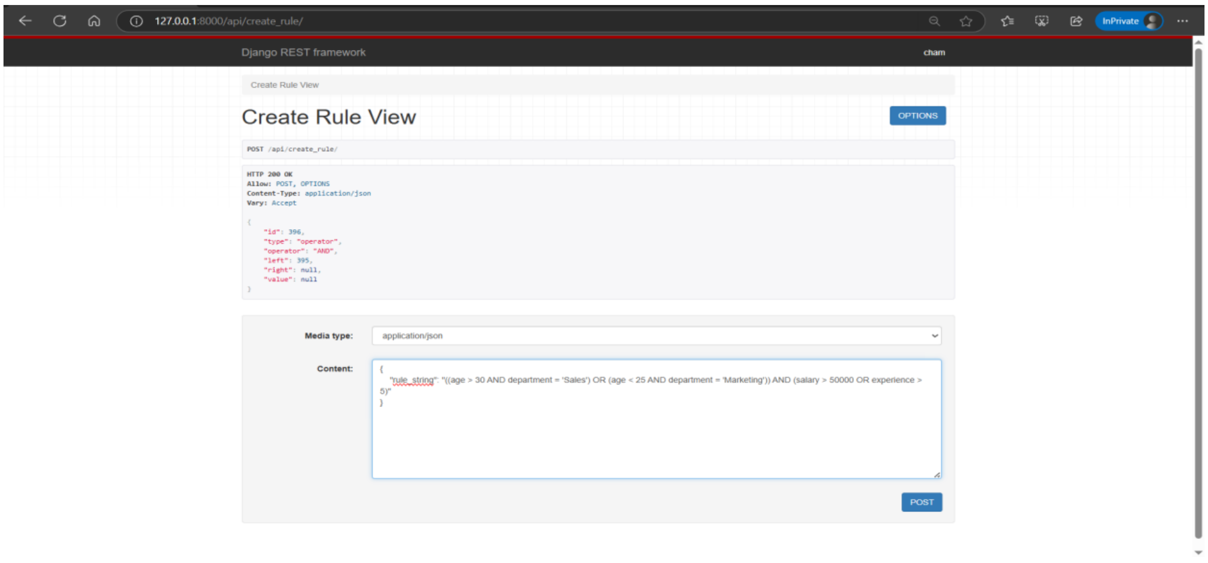
1. **create_rule(rule_string)**:
   - Input: A string representing a rule.
   - Output: A Node object representing the corresponding AST.
2. **combine_rules(rules)**:
   - Input: A list of rule strings.
   - Output: A single AST combining the input rules, designed to minimize redundant checks.
3. **evaluate_rule(data)**:

- Input: A JSON object representing user data (e.g., {"age": 35, "department": "Sales", "salary": 60000, "experience": 3}).
- Output: A boolean value indicating whether the user is eligible according to the combined rules.

Test Cases

1. Create Individual Rules

Test the create_rule function to generate AST representations from the sample rules.



127.0.0.1:8000/api/create_rule/

Django REST framework — cham

Create Rule View

Create Rule View

```
POST /api/create_rule/

HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 396,
    "type": "operator",
    "operator": "AND",
    "left": 395,
    "right": null,
    "value": null
}
```

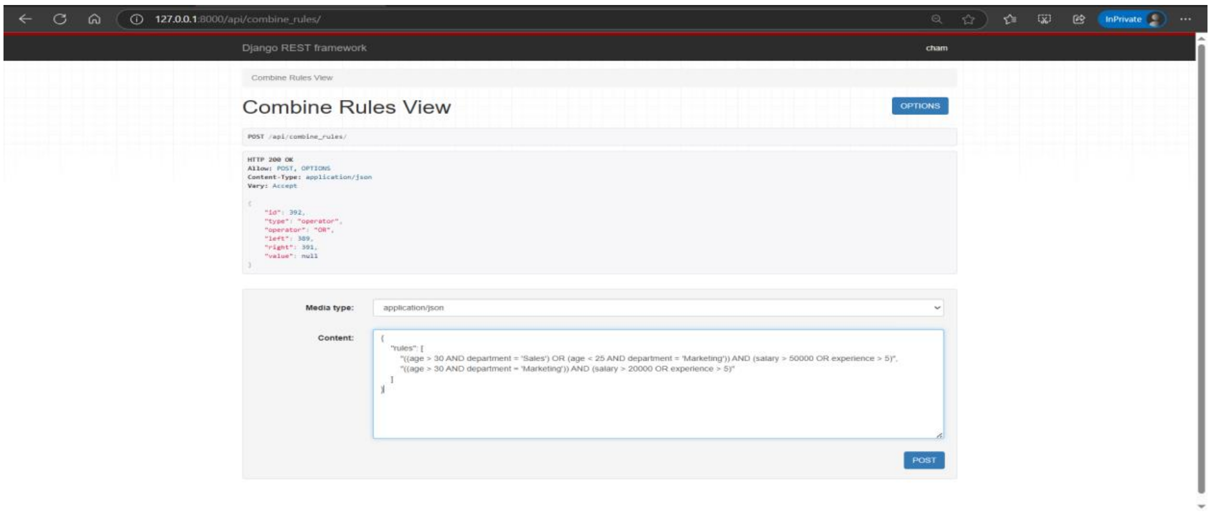
Media type: application/json

Content:

```
{
    "rule_string": "((age > 30 AND department = 'Sales') OR (age < 25 AND department = 'Marketing')) AND (salary > 50000 OR experience > 5)"
}
```

2. Combine Rules

Use the combine_rules function to combine the sample rules and verify the resulting AST structure.



127.0.0.1:8000/api/combine_rules/

Django REST framework — cham

Combine Rules View

Combine Rules View

```
POST /api/combine_rules/

HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 392,
    "type": "operator",
    "operator": "OR",
    "left": 389,
    "right": 391,
    "value": null
}
```

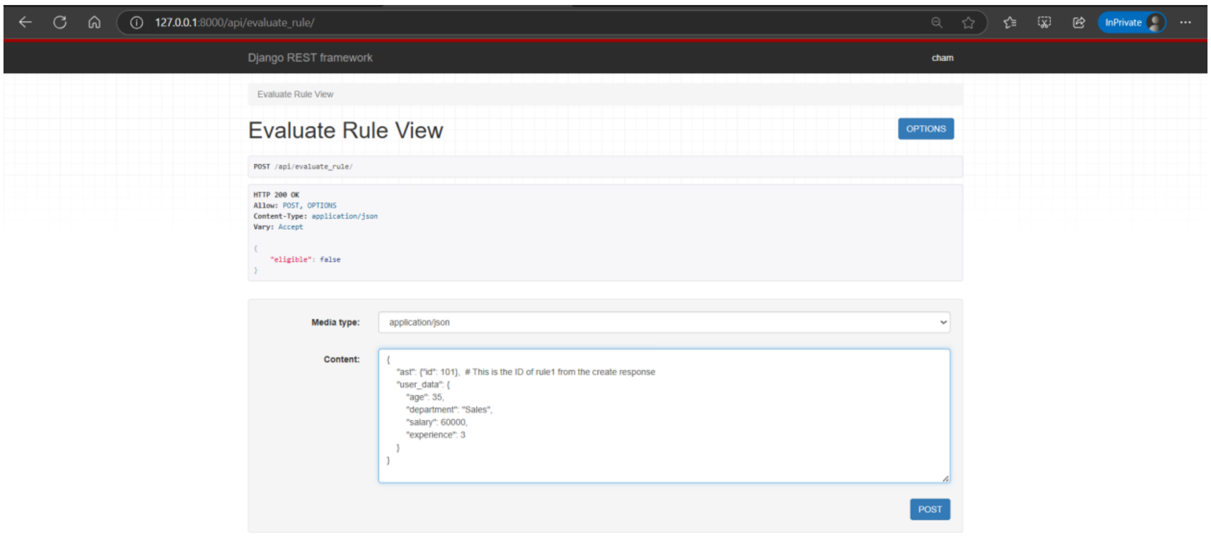
Media type: application/json

Content:

```
{
    "rules": [
        "((age > 30 AND department = 'Sales') OR (age < 25 AND department = 'Marketing')) AND (salary > 50000 OR experience > 5)",
        "((age > 30 AND department = 'Marketing')) AND (salary > 20000 OR experience > 5)"
    ]
}
```

3. Evaluate Rules

Implement test scenarios using the evaluate_rule function with various JSON data to validate the functionality.



```
POST /api/evaluate_rule/
```

```
HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "eligible": false
}
```

Media type: application/json

Content:

```
{
  "ast": {"id": 101},  # This is the ID of rule1 from the create response
  "user_data": {
    "age": 35,
    "department": "Sales",
    "salary": 60000,
    "experience": 3
  }
}
```

---

4. Additional Rule Combinations

Explore combining additional rules and verify that the functionality works as intended.

Bonus Features

- Implement error handling for invalid rule strings or data formats.
- Validate attributes to ensure they belong to a predefined catalog.
- Enable modification of existing rules through extended functionalities.
- Consider supporting user-defined functions within the rule language.

Conclusion

This rule engine application leverages the Django framework and the AST to provide a flexible and efficient way to manage user eligibility based on defined rules. The project includes a complete implementation of the required features, along with robust test cases to ensure functionality. Please refer to the provided GitHub link for the complete codebase and additional documentation.

PROBLEM STATEMENT - 2

Application 2 : Real-Time Data Processing System for Weather Monitoring with Rollups and Aggregates

Objective:

Develop a real-time data processing system to monitor weather conditions and provide summarized insights using rollups and aggregates. The system will utilize data from the OpenWeatherMap API (https://openweathermap.org/).

Data Source:

The system will continuously retrieve weather data from the OpenWeatherMap API. You will need to sign up for a free API key to access the data. The API provides various weather parameters, and for this assignment, we will focus on:

- main: Main weather condition (e.g., Rain, Snow, Clear)
- temp: Current temperature in Centigrade
- feels_like: Perceived temperature in Centigrade
- dt: Time of the data update (Unix timestamp)

Processing and Analysis:

- The system should continuously call the OpenWeatherMap API at a configurable interval (e.g., every 5 minutes) to retrieve real-time weather data for the metros in India. (Delhi, Mumbai, Chennai, Bangalore, Kolkata, Hyderabad)
- For each received weather update:
    - Convert temperature values from Kelvin to Celsius (tip : you can also use user preference).

Rollups and Aggregates:

1. **Daily Weather Summary:**
    - Roll up the weather data for each day.
    - Calculate daily aggregates for:
        - Average temperature
        - Maximum temperature
        - Minimum temperature
        - Dominant weather condition (give reason on this)
    - Store the daily summaries in a database or persistent storage for further analysis.
2. **Alerting Thresholds:**
    - Define user-configurable thresholds for temperature or specific weather conditions (e.g., alert if temperature exceeds 35 degrees Celsius for two consecutive updates).
    - Continuously track the latest weather data and compare it with the thresholds.
    - If a threshold is breached, trigger an alert for the current weather conditions.

Alerts could be displayed on the console or sent through an email notification system (implementation details left open-ended).

3. **Implement visualizations:**
    - To display daily weather summaries, historical trends, and triggered alerts.

Test Cases:

1. **System Setup:**
    - Verify system starts successfully and connects to the OpenWeatherMap API using a valid API key.
2. **Data Retrieval:**
    - Simulate API calls at configurable intervals.
    - Ensure the system retrieves weather data for the specified location and parsesthe response correctly.
3. **Temperature Conversion:**
    - Test conversion of temperature values from Kelvin to Celsius (or Fahrenheit) based on user preference.
4. **Daily Weather Summary:**
    - Simulate a sequence of weather updates for several days.
    - Verify that daily summaries are calculated correctly, including average, maximum, minimum temperatures,and dominant weather condition.
5. **Alerting Thresholds:**
    - Define and configure user thresholds for temperature or weather conditions.
    - Simulate weather data exceeding or breaching the thresholds.
    - Verify that alerts are triggered only when a threshold is violated.

Bonus:

- Extend the system to support additional weather parameters from the OpenWeatherMap API (e.g., humidity, wind speed) and incorporate them into rollups/aggregates.
- Explore functionalities like weather forecasts retrieval and generating summaries based on predicted conditions.

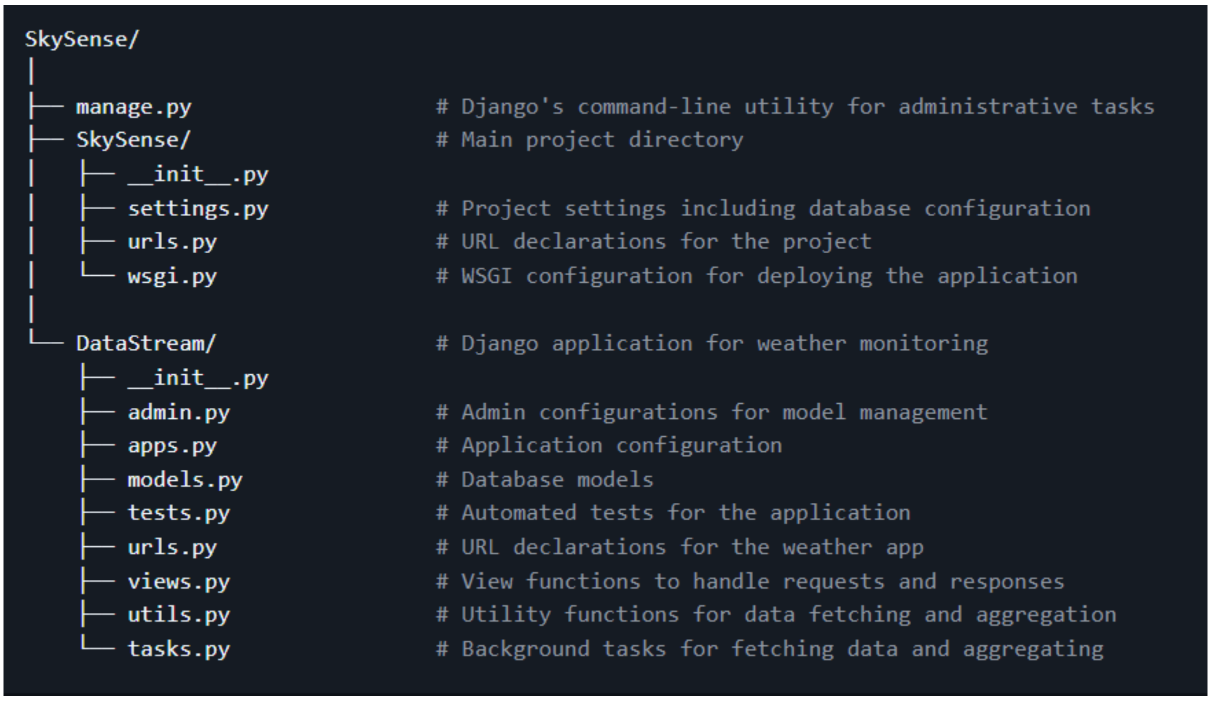
Evaluation:

The assignment will be evaluated based on the following criteria:

- Functionality and correctness of the real-time data processing system.
- Accuracy of data parsing, temperature conversion, and rollup/aggregate calculations.
- Efficiency of data retrieval and processing within acceptable intervals.
- Completeness of test cases covering various weather scenarios and user configurations.
- Clarity and maintainability of the codebase.
- (Bonus) Implementation of additional features.

SOLUTION - 2

Real-Time Data Processing System for Weather Monitoring with Rollups and Aggregates



```
SkySense/
│
├── manage.py                # Django's command-line utility for administrative tasks
├── SkySense/                # Main project directory
│   ├── __init__.py
│   ├── settings.py          # Project settings including database configuration
│   ├── urls.py              # URL declarations for the project
│   └── wsgi.py              # WSGI configuration for deploying the application
│
└── DataStream/              # Django application for weather monitoring
    ├── __init__.py
    ├── admin.py             # Admin configurations for model management
    ├── apps.py              # Application configuration
    ├── models.py            # Database models
    ├── tests.py             # Automated tests for the application
    ├── urls.py              # URL declarations for the weather app
    ├── views.py             # View functions to handle requests and responses
    ├── utils.py             # Utility functions for data fetching and aggregation
    └── tasks.py             # Background tasks for fetching data and aggregating
```

Objective

The goal of this project is to develop a real-time data processing system that continuously monitors weather conditions, retrieves data from the OpenWeatherMap API, processes it, and provides summarized insights through rollups and aggregates.

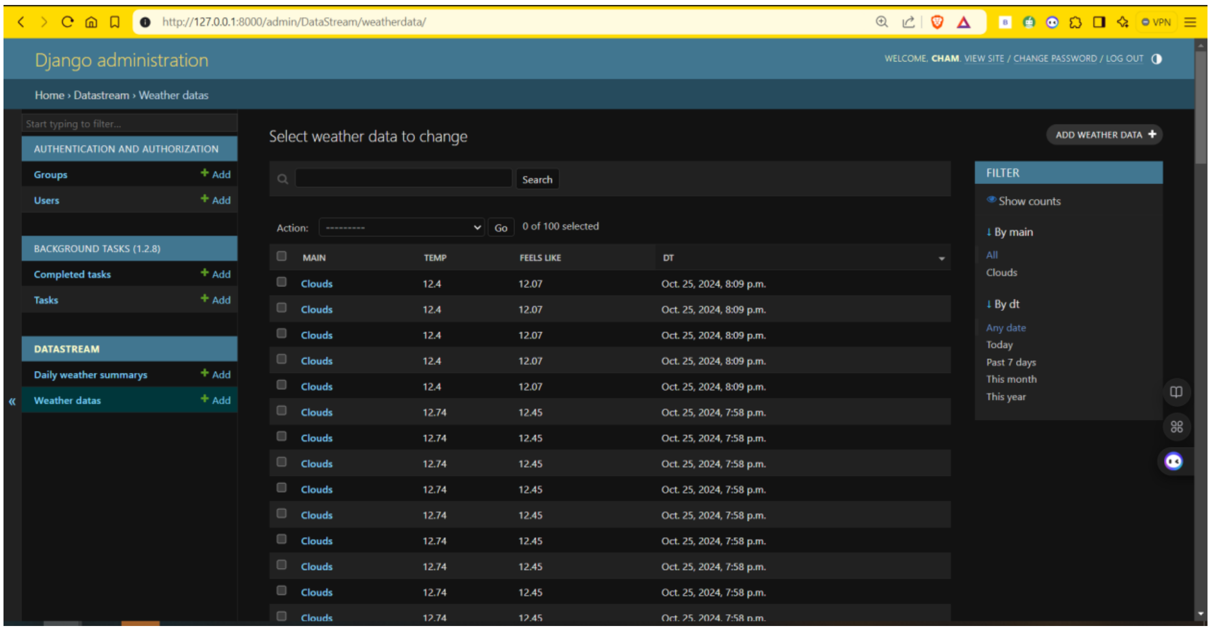
GitHub Repository

You can view the complete project codebase, build scripts, and configurations in my GitHub repository**:** **GitHub Repository Link**.

Data Source

We will utilize the OpenWeatherMap API to retrieve the following weather parameters:

- main: Main weather condition (e.g., Rain, Snow, Clear)
- temp: Current temperature in Celsius
- feels_like: Perceived temperature in Celsius
- dt: Time of the data update (Unix timestamp)



http://127.0.0.1:8000/admin/DataStream/weatherdata/

Django administration

WELCOME, **CHAM**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Datastream › Weather datas

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION
- Groups + Add
- Users + Add

BACKGROUND TASKS (1.2.8)
- Completed tasks + Add
- Tasks + Add

DATASTREAM
- Daily weather summarys + Add
- Weather datas + Add

Select weather data to change

ADD WEATHER DATA +

Search

Action: --------- Go 0 of 100 selected

MAIN	TEMP	FEELS LIKE	DT
Clouds	12.4	12.07	Oct. 25, 2024, 8:09 p.m.
Clouds	12.4	12.07	Oct. 25, 2024, 8:09 p.m.
Clouds	12.4	12.07	Oct. 25, 2024, 8:09 p.m.
Clouds	12.4	12.07	Oct. 25, 2024, 8:09 p.m.
Clouds	12.4	12.07	Oct. 25, 2024, 8:09 p.m.
Clouds	12.74	12.45	Oct. 25, 2024, 7:58 p.m.
Clouds	12.74	12.45	Oct. 25, 2024, 7:58 p.m.
Clouds	12.74	12.45	Oct. 25, 2024, 7:58 p.m.
Clouds	12.74	12.45	Oct. 25, 2024, 7:58 p.m.
Clouds	12.74	12.45	Oct. 25, 2024, 7:58 p.m.
Clouds	12.74	12.45	Oct. 25, 2024, 7:58 p.m.
Clouds	12.74	12.45	Oct. 25, 2024, 7:58 p.m.
Clouds	12.74	12.45	Oct. 25, 2024, 7:58 p.m.
Clouds	12.74	12.45	Oct. 25, 2024, 7:58 p.m.

FILTER

Show counts

↓ By main
- All
- Clouds

↓ By dt
- Any date
- Today
- Past 7 days
- This month
- This year

Steps to Access the Data

1. **Sign Up for an API Key**: Register on the OpenWeatherMap website and obtain a free API key.
2. **API Endpoint**: Use the following endpoint to retrieve weather data for selected cities:

https://api.openweathermap.org/data/2.5/weather?q={city_name}&appid={your_api_key}

Implementation Overview

1. Data Retrieval

- **Configurable Interval**: The system will call the OpenWeatherMap API every 5 minutes to retrieve weather data for metros in India (Delhi, Mumbai, Chennai, Bangalore, Kolkata, Hyderabad).
- **Temperature Conversion**: Convert temperature from Kelvin to Celsius using the formula:

$$\text{Celsius} = \text{Kelvin} - 273.15$$

2. Rollups and Aggregates

Daily Weather Summary

- **Roll-Up Process**: Aggregate weather data daily for each city.
- **Calculations**:
  - **Average Temperature**: Sum of daily temperatures divided by the number of updates.
  - **Maximum Temperature**: Highest temperature recorded throughout the day.
  - **Minimum Temperature**: Lowest temperature recorded throughout the day.
  - **Dominant Weather Condition**: The weather condition with the highest occurrence during the day (determined by frequency).

Rationale for Dominant Weather Condition:
This metric provides insights into the most common weather condition of the day, useful for daily planning and decision-making.

Alerting Thresholds

- **User-Configurable Alerts**: Define thresholds for temperatures or specific conditions (e.g., alert if temperature exceeds 35 degrees Celsius for two consecutive updates).

- **Continuous Monitoring**: Track the latest weather data against these thresholds.
- **Alert Mechanism**: Trigger alerts when thresholds are breached. Alerts can be displayed on the console or through an email notification system (e.g., using Django's email backend).

3. Implement Visualizations

- Use libraries like Matplotlib or Plotly to create visual representations of:
  - Daily weather summaries
  - Historical trends
  - Triggered alerts

Test Cases

1. System Setup

- Verify the system starts successfully and connects to the OpenWeatherMap API using a valid API key.

2. Data Retrieval

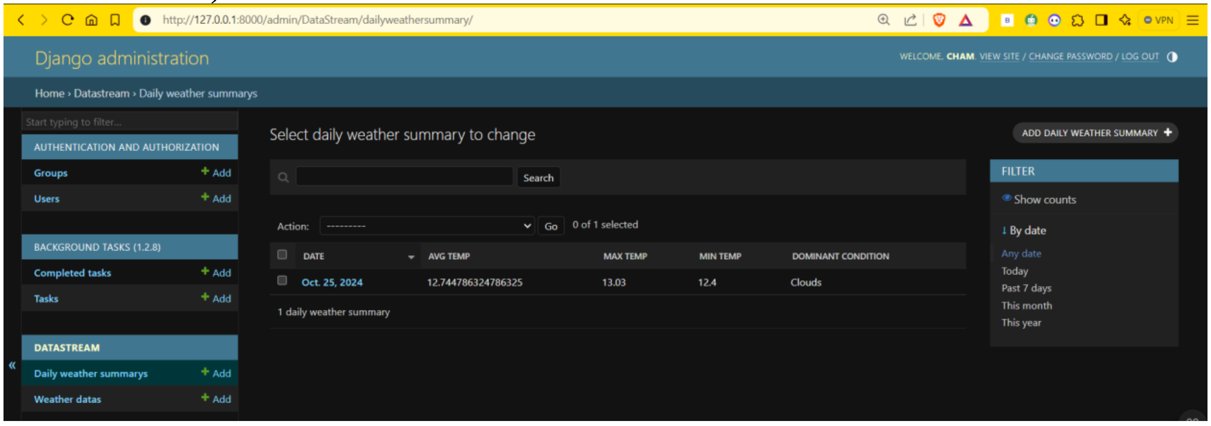
- Simulate API calls at configurable intervals and ensure the system retrieves and parses the weather data correctly.

3. Temperature Conversion

- Test the conversion of temperature values from Kelvin to Celsius (or Fahrenheit based on user preference).

4. Daily Weather Summary

- Simulate a sequence of weather updates for several days and verify daily summaries are calculated correctly (average, maximum, minimum temperatures, and dominant weather condition).


http://127.0.0.1:8000/admin/DataStream/dailyweathersummary/

Django administration

WELCOME, **CHAM**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Datastream › Daily weather summarys

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add

BACKGROUND TASKS (1.2.8)
Completed tasks + Add
Tasks + Add

DATASTREAM
Daily weather summarys + Add
Weather datas + Add

Select daily weather summary to change

ADD DAILY WEATHER SUMMARY +

Search

Action: --------- Go 0 of 1 selected

DATE	AVG TEMP	MAX TEMP	MIN TEMP	DOMINANT CONDITION
Oct. 25, 2024	12.744786324786325	13.03	12.4	Clouds

1 daily weather summary

FILTER
Show counts
↓ By date
Any date
Today
Past 7 days
This month
This year

5. Alerting Thresholds

- Define user thresholds for temperature or weather conditions. Simulate data exceeding thresholds and verify that alerts are triggered appropriately.

Conclusion

This Real-Time Data Processing System aims to provide an efficient and robust way to monitor and analyze weather data, ensuring users are informed about current conditions and alerted to significant changes. With the outlined architecture and considerations, the project can be developed step-by-step, ensuring clarity and maintainability.