Project Report
Milestone 3

Manik Jain
(2022MCS2832)

Sagar Agrawal
(2022MCS2065)

Vinit Chandak
(2022EET2109)

02/05/2023

—

Introduction to Database Systems

(COL 362/COL 632)

—

Prof. Srikanta Bedathur

Overall Architecture

We've created a webapp that can be used following:
1. Visualization of different weather parameters like temperature, rainfall, etc. for different cities in India.
2. Current weather
3. Weather Forecast
4. Weather Alerts

When the server is started, the user would be on the landing page. On this page the user can select one of the parameters to visualize the data for that, sign up for alerts or see the current weather and forecasts. Once the user makes their selection, they are directed to the corresponding page.

For the weather parameters, once on the page, the user can select the states, cities, subdivisions, date(range or single), different attributes from the available options for each parameter after which the selections are appropriately visualized. For each attribute selected, a graph will be plotted in a new tab.

Next up, on the landing page, there is an option to see the current weather and weather forecasts. Once the user clicks on it, they are redirected to a page where they can select a state. After that they can select a city. Then the current weather details of the selected city are shown. Below that, there is dropdown from which the user can select the granularity of forecasts that they want.

Also, on the landing there is an option for weather alerts. Once the user clicks on that, they are redirected to new page. On this page the user can sign up for weather alerts or can deregister from the same.

The dropdowns are populated by running appropriate SQL queries in the backend.
The data for current weather details, forecasts and alerts are fetched from OpenWeatherMap API. Ideally this should be done on the basis of database but since we could only find outdated data, we're fetching these from the API.
Before the demo, we might make some changes to the UI and add new features if the time permits.

We've used flask along with SQLAlchemy for the implementation along with some helper libraries.

Application Front End Design and Queries/Transactions

Landing page:
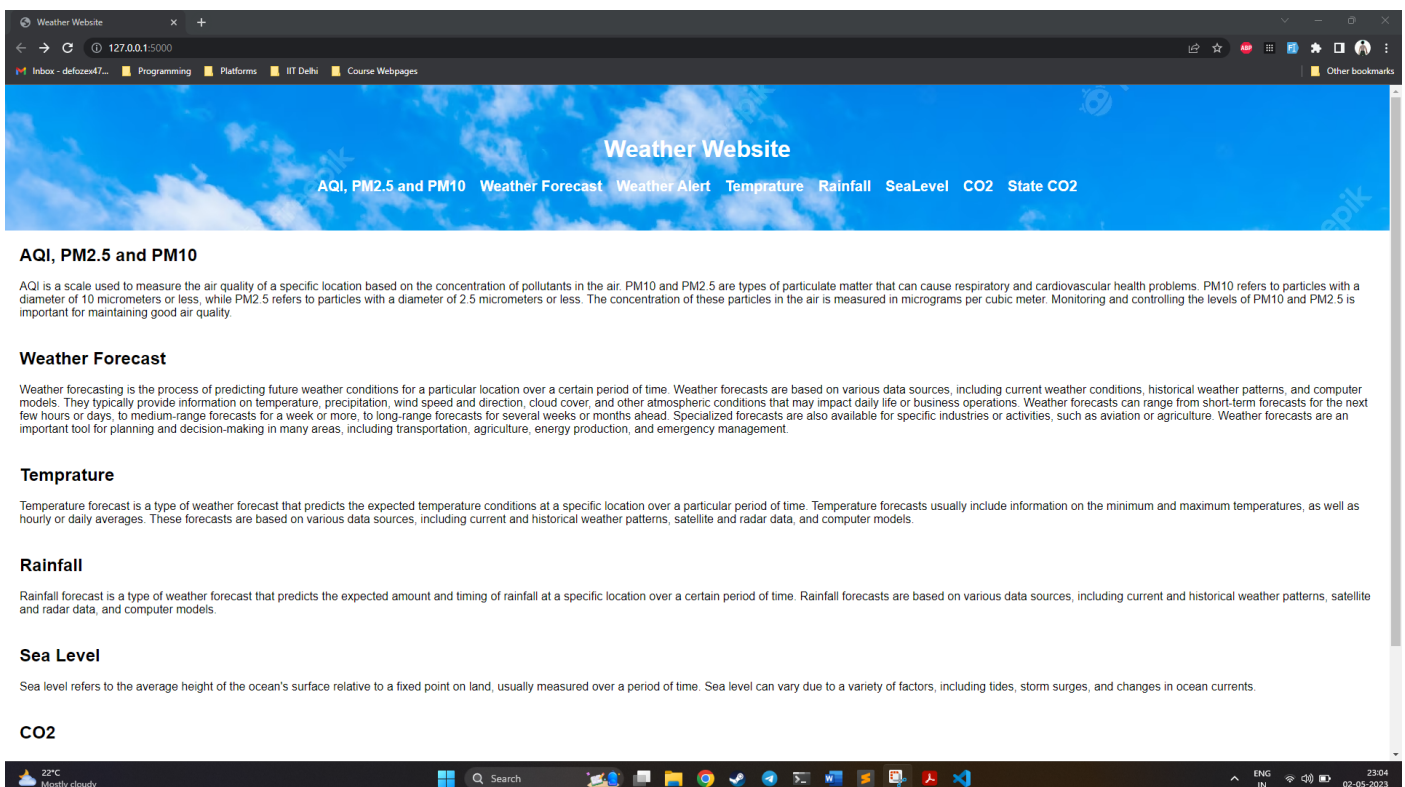
Once the user makes a selection, they are directed to the appropriate page. For example, we'll select AQI. The user is then redirected to this page. Firstly, the dropdown for selection of a state is populated based on the data in the AQI table. The following query is used for this purpose:

```
SELECT DISTINCT State FROM AQI;
```

The based on the selected state(s), the dropdown for selection of a city is populated with the help of the following query:

```
SELECT DISTINCT City FROM AQI WHERE State='Delhi' OR state = "Maharashtra";
```

Then the user selects a single date or a range of dates. One of the following queries is ran in the background after this selection and all the rows which satisfy all the conditions are fetched from the AQI table.

```sql
    --For dropdown--
      SELECT * FROM AQI WHERE State ='Gujarat';
      SELECT * FROM AQI WHERE State='Gujarat' AND City='Ahmedabad';
      SELECT * FROM AQI WHERE State='Gujarat' AND City='Ahmedabad' AND EXTRACT(YEAR FROM
DATE)='2015';
      SELECT * FROM AQI WHERE State='Gujarat' AND City='Ahmedabad' AND EXTRACT(YEAR FROM
DATE)='2015' AND EXTRACT(MONTH FROM DATE)='01';
      SELECT * FROM AQI WHERE State='Gujarat' AND City='Ahmedabad' AND EXTRACT(YEAR FROM
DATE)='2015' AND EXTRACT(MONTH FROM DATE)='01' AND EXTRACT(DAY FROM DATE)='01';
    --For dropdown and range--
      SELECT * FROM AQI WHERE State='Gujarat' AND City='Ahmedabad' AND EXTRACT(YEAR FROM
DATE) BETWEEN '2015' AND '2016';
      SELECT * FROM AQI WHERE State='Gujarat' AND City='Ahmedabad' AND EXTRACT(YEAR FROM
DATE)='2015' AND EXTRACT(MONTH FROM DATE) BETWEEN '01' AND '03';
      SELECT * FROM AQI WHERE State='Gujarat' AND City='Ahmedabad' AND EXTRACT(YEAR FROM
DATE)='2015' AND EXTRACT(MONTH FROM DATE)='01' AND EXTRACT(DAY FROM DATE) BETWEEN '1' AND
'15';
```

The user then selects the attributes for which they want to visualize the data. For each attribute, we get a separate graph in a new tab.
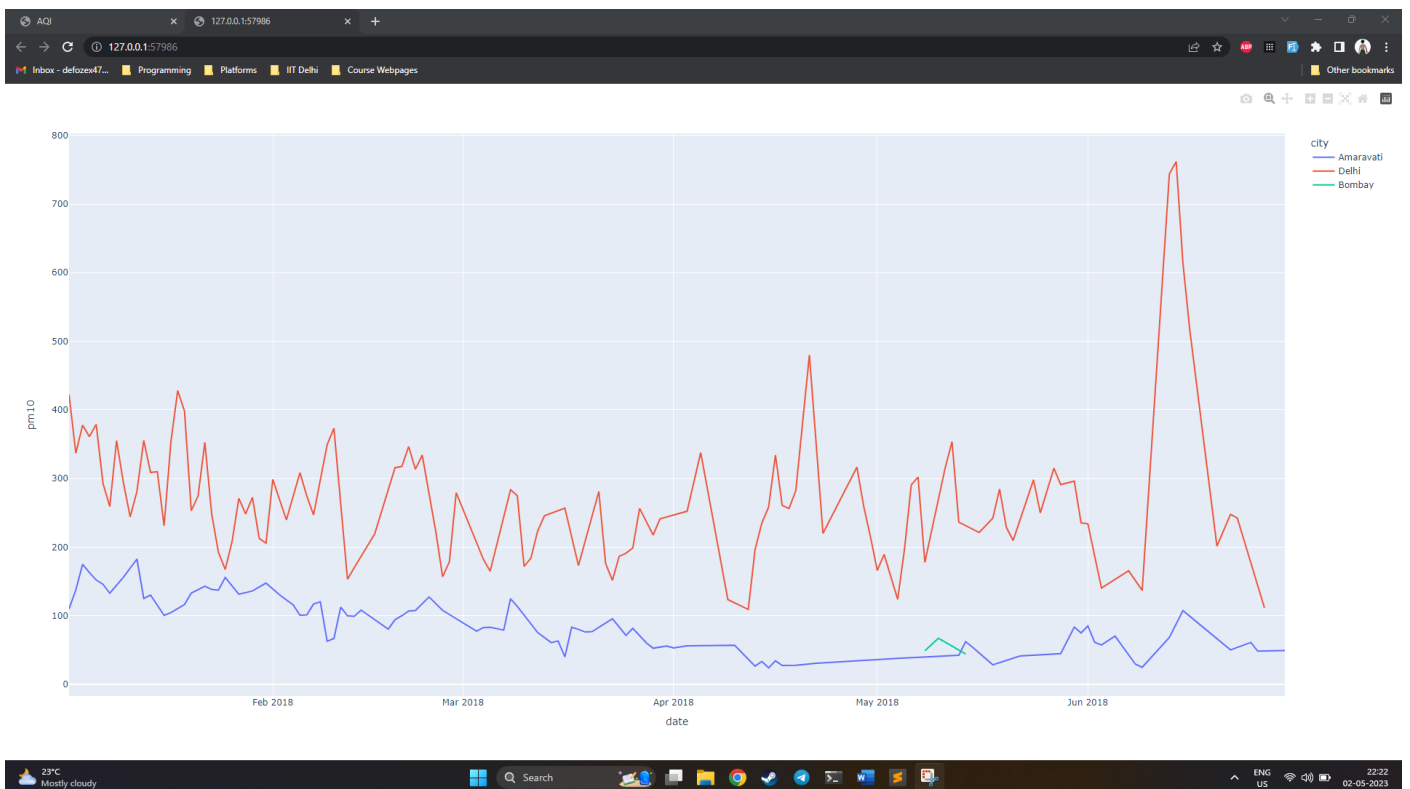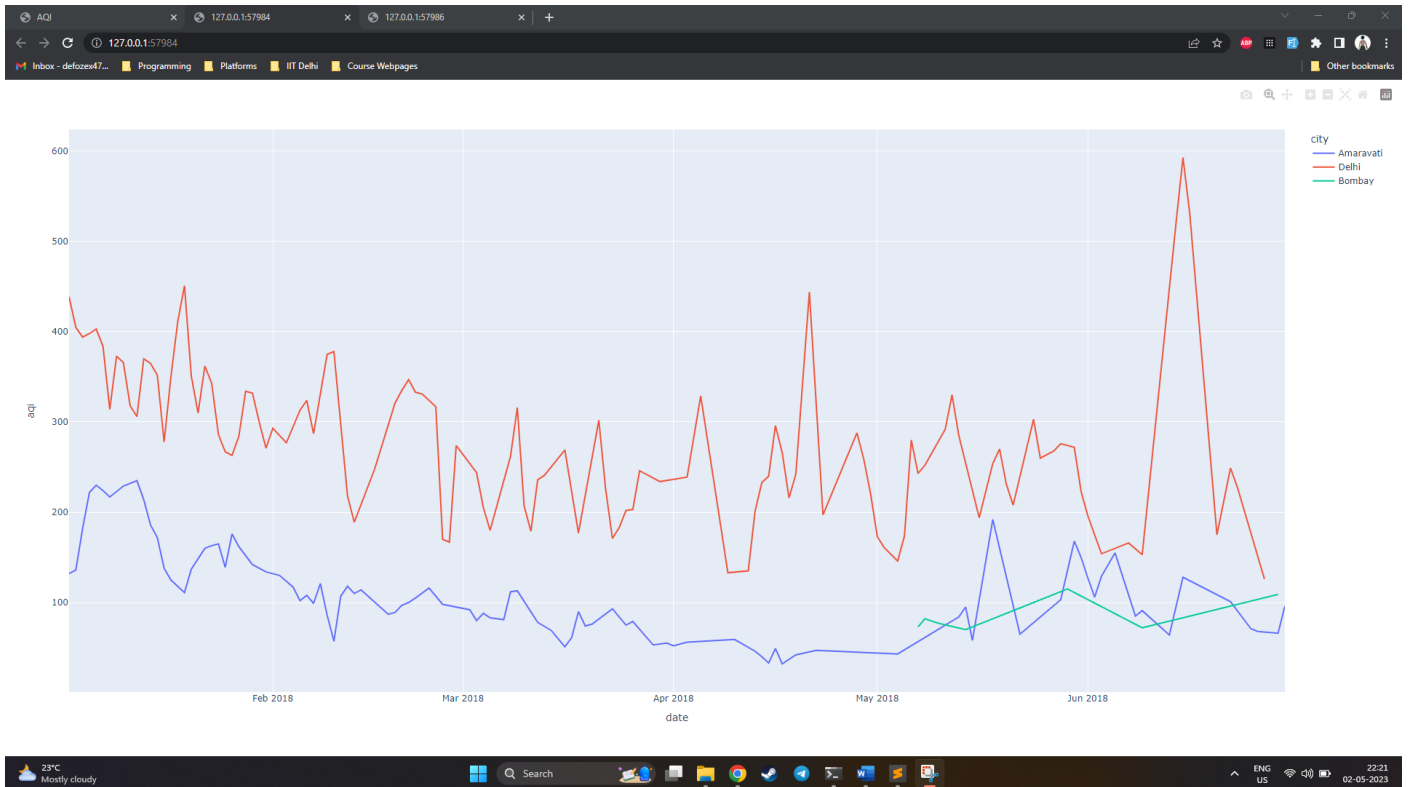
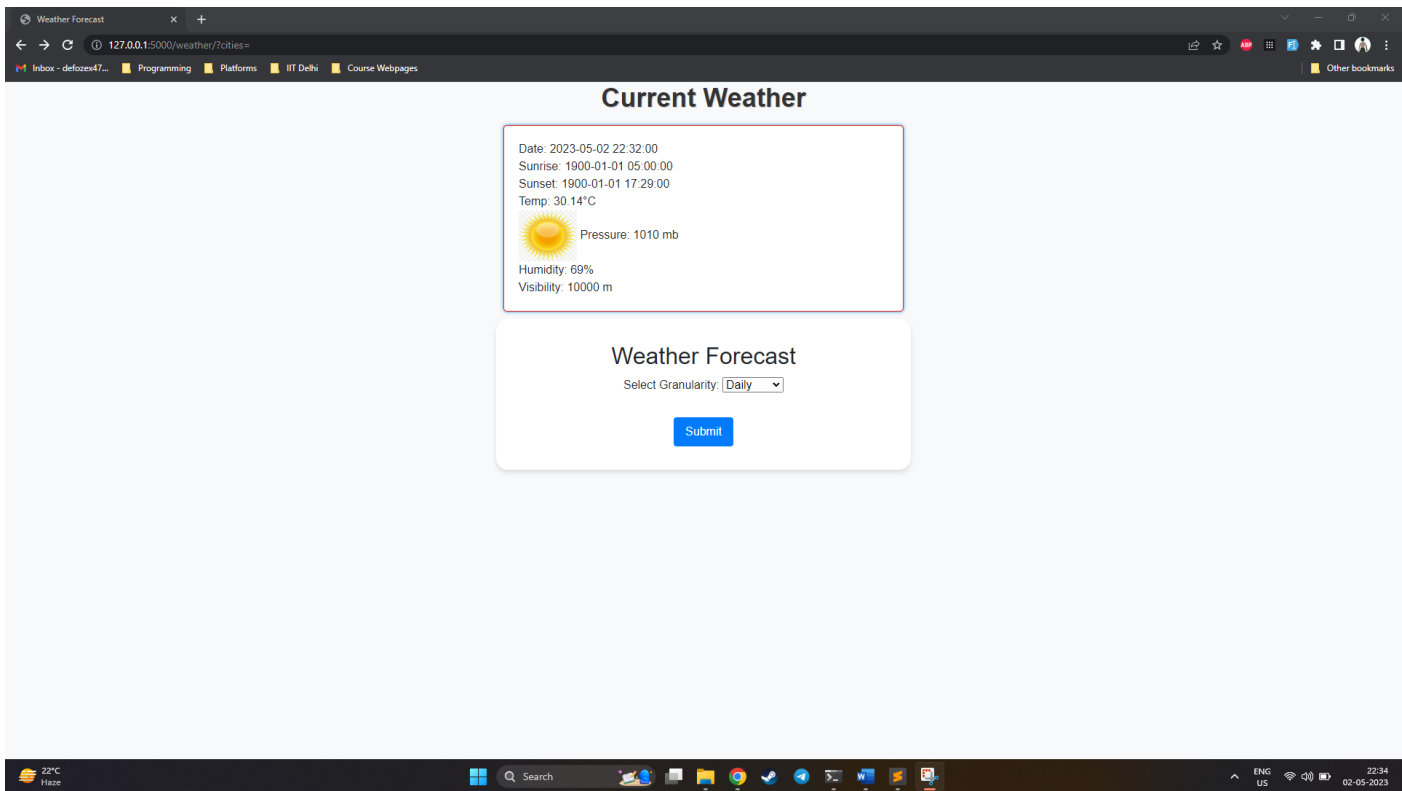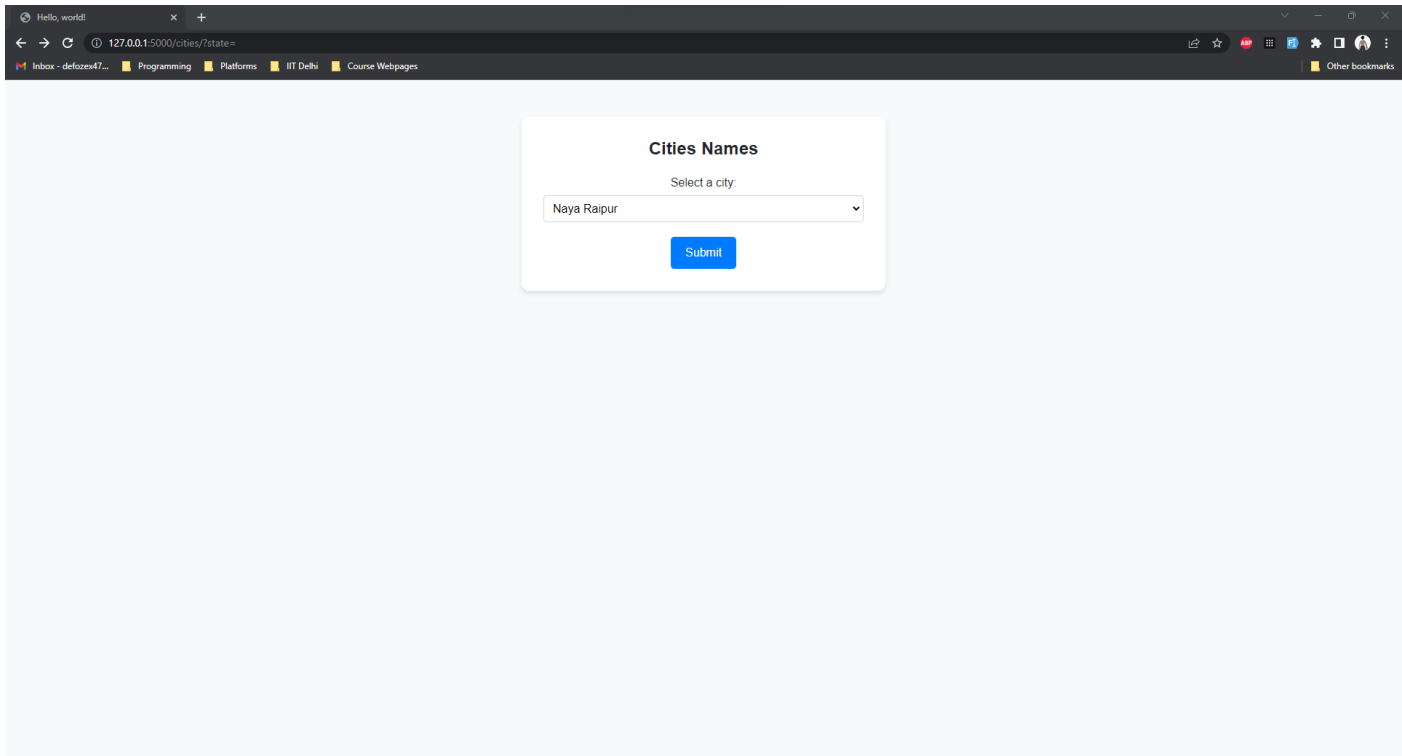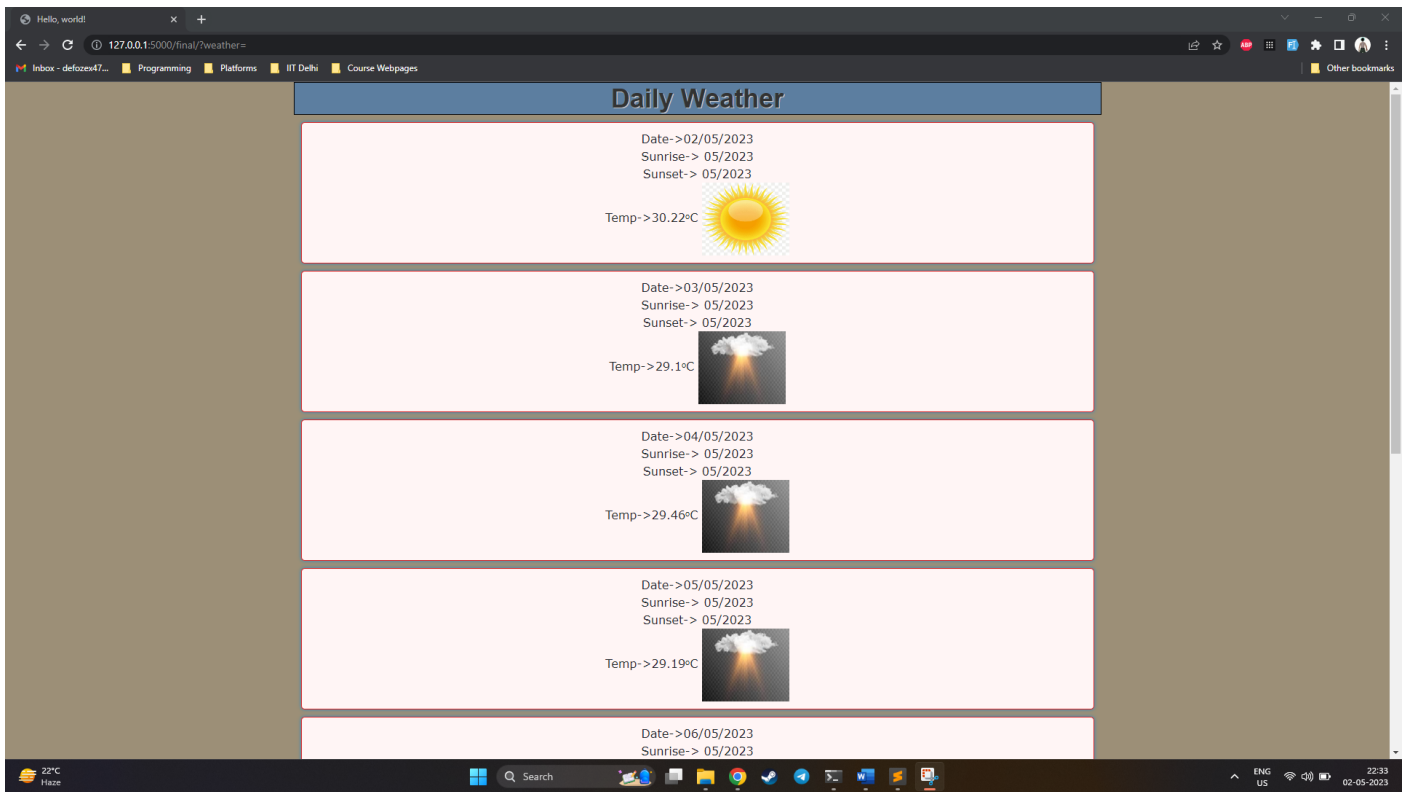Current weather and forecasts:

Once the user selects a state and a city, they are shown the current weather for that region. Below that they have the options to see weather forecasts for that region with different granularities. The daily forecasts are also stored in the database.
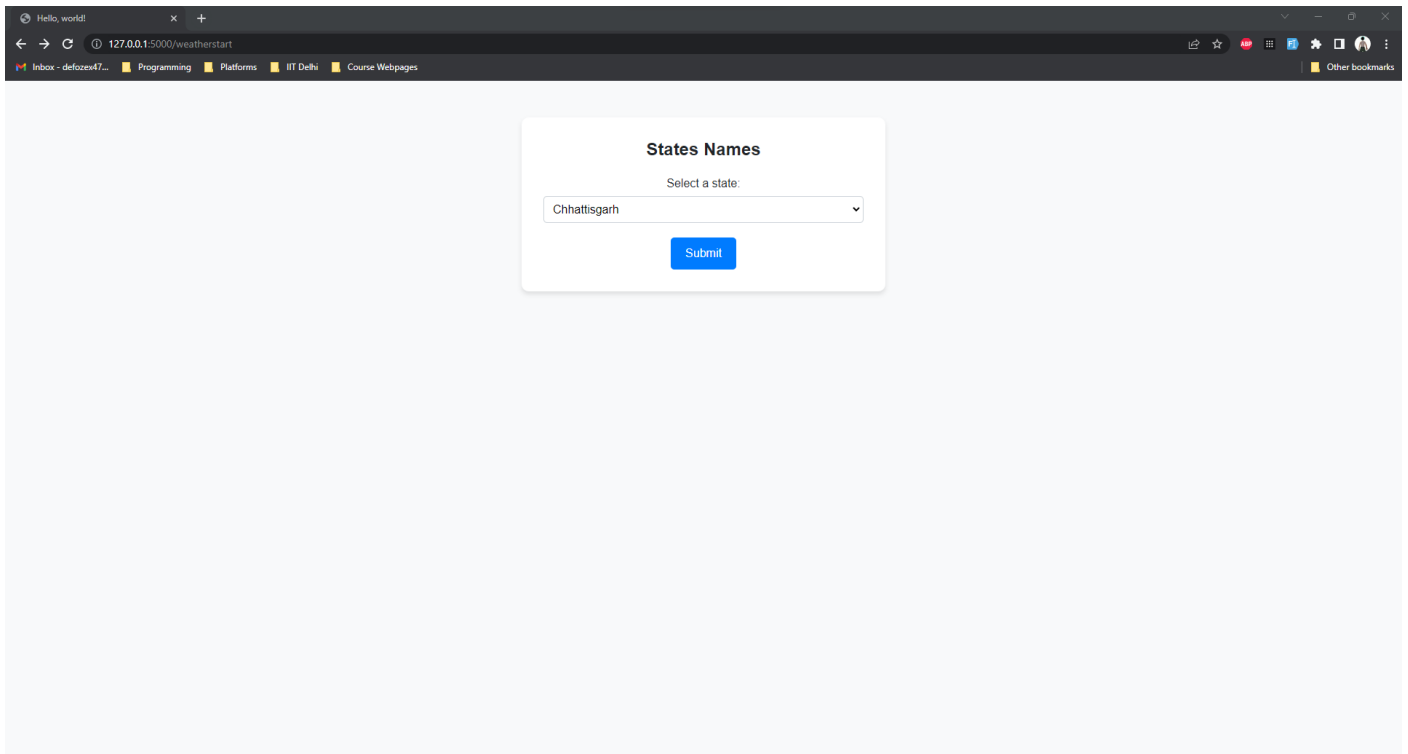
Weather Alerts:

Once a user signs up for alerts, their details are stored in the database. Then everytime the server is started, a procedure calls the OpenWeatherMap API for every entry in the "weather_alerts" table. If there is an alert in the area, a email is sent to the user with the details of the alert.
The user can opt out of the alerts through the same page. On opting out, the users details are deleted from the database.
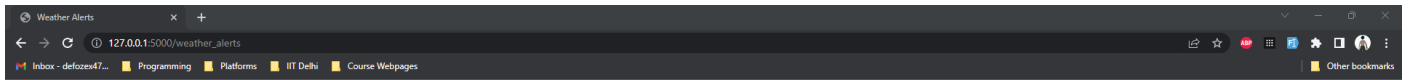
Database Features Implemented

1. Deferred foreign key constraints on states and cities mitigating the circular dependency.
2. ON DELETE CASCADE on all the foreign keys.
3. Indexes
4. Trigger for data integrity.

   Ensuring that a new entry in the Average_Temperature table has a valid temperature value.

```sql
CREATE OR REPLACE FUNCTION check_valid_temperature()
RETURNS TRIGGER AS $$
BEGIN
IF NEW.Average_Temperature < -273.15 THEN
    RAISE EXCEPTION 'Invalid temperature value: below absolute zero.';
END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER trg_check_valid_temperature
BEFORE INSERT OR UPDATE ON Average_Temperature
FOR EACH ROW
EXECUTE PROCEDURE check_valid_temperature();
```

5. Trigger for cascading actions.

   If the capital of a state is updated, then it is also updated in all the relevent tables.

```sql
CREATE OR REPLACE FUNCTION update_state_capital()
RETURNS TRIGGER AS $$
BEGIN
-- Check if the capital has changed
IF OLD.Capital <> NEW.Capital THEN
    -- Update the capital in the Cities table
    UPDATE Cities
    SET City = NEW.Capital
    WHERE City = OLD.Capital
    AND State = NEW.State;

    -- Update the capital in the Sealevel table
    UPDATE Sealevel
    SET Sea_Shore_City = NEW.Capital
```

```sql
        WHERE Sea_Shore_City = OLD.Capital
        AND State = NEW.State;
END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_state_capital
AFTER UPDATE ON States
FOR EACH ROW
EXECUTE PROCEDURE update_state_capital();
```

6. Trigger for auto updates.

   Automatically calculating the Annual and quarterly rainfall in the Rainfall
   table based on the monthly values.

```sql
CREATE OR REPLACE FUNCTION calculate_annual_quarterly_rainfall()
RETURNS TRIGGER AS $$
BEGIN
    -- Calculate Annual rainfall
    NEW.Annual = COALESCE(NEW.January, 0) + COALESCE(NEW.February, 0) +
COALESCE(NEW.March, 0) + COALESCE(NEW.April, 0)
                + COALESCE(NEW.May, 0) + COALESCE(NEW.June, 0) + COALESCE(NEW.July, 0) +
COALESCE(NEW.August, 0)
                + COALESCE(NEW.September, 0) + COALESCE(NEW.October, 0) +
COALESCE(NEW.November, 0) + COALESCE(NEW.December, 0);

    -- Calculate quarterly rainfall
    NEW.January_February = COALESCE(NEW.January, 0) + COALESCE(NEW.February, 0);
    NEW.March_May = COALESCE(NEW.March, 0) + COALESCE(NEW.April, 0) + COALESCE(NEW.May,
0);
    NEW.June_September = COALESCE(NEW.June, 0) + COALESCE(NEW.July, 0) +
COALESCE(NEW.August, 0) + COALESCE(NEW.September, 0);
    NEW.October_December = COALESCE(NEW.October, 0) + COALESCE(NEW.November, 0) +
COALESCE(NEW.December, 0);

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_calculate_annual_quarterly_rainfall
BEFORE INSERT OR UPDATE ON Rainfall
FOR EACH ROW
EXECUTE PROCEDURE calculate_annual_quarterly_rainfall();
```

7. Logging - Created a log table which tracks changes in the average_temperature table.
8. Trigger for logging.

Created a trigger to log changes in the average_temperature table.

```sql
CREATE OR REPLACE FUNCTION average_temperature_audit_trigger_func()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'UPDATE' THEN
        INSERT INTO average_temperature_audit (
            operation, state, month, year,
            old_avg_temperature, new_avg_temperature,
            old_avg_temperature_uncertainty, new_avg_temperature_uncertainty,
            changed_by, changed_at
        )
        VALUES (
            'UPDATE', NEW.state, NEW.month, NEW.year,
            OLD.average_temperature, NEW.average_temperature,
            OLD.average_temperature_uncertainty, NEW.average_temperature_uncertainty,
            current_user, current_timestamp
        );
        RETURN NEW;
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO average_temperature_audit (
            operation, state, month, year,
            old_avg_temperature, new_avg_temperature,
            old_avg_temperature_uncertainty, new_avg_temperature_uncertainty,
            changed_by, changed_at
        )
        VALUES (
            'DELETE', OLD.state, OLD.month, OLD.year,
            OLD.average_temperature, NULL,
            OLD.average_temperature_uncertainty, NULL,
            current_user, current_timestamp
        );
        RETURN OLD;
    ELSIF TG_OP = 'INSERT' THEN
        INSERT INTO average_temperature_audit (
            operation, state, month, year,
            old_avg_temperature, new_avg_temperature,
            old_avg_temperature_uncertainty, new_avg_temperature_uncertainty,
            changed_by, changed_at
        )
        VALUES (
            'INSERT', NEW.state, NEW.month, NEW.year,
            NULL, NEW.average_temperature,
```

```sql
            NULL, NEW.average_temperature_uncertainty,
            current_user, current_timestamp
        );
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER average_temperature_audit_trigger
AFTER INSERT OR UPDATE OR DELETE ON average_temperature
FOR EACH ROW
EXECUTE PROCEDURE average_temperature_audit_trigger_func();
```

9. Materialized Views

Created a materialized view to display the total CO2 emissions for each state.

```sql
CREATE MATERIALIZED VIEW total_state_co2_emissions AS
SELECT State, SUM(CO2_Emissions) as total_co2_emissions
FROM State_CO2
GROUP BY State;
```

Created a materialized view that'll keep tab on all the cities where aqi quality bucket has been "Very Poor" in the past decade.

```sql
CREATE MATERIALIZED VIEW severe_aqi_cities_past_decade AS
SELECT
    City,
    State,
    COUNT(*) AS num_severe_days
FROM
    AQI
WHERE
    AQI_Bucket = 'Very Poor'
    AND Date >= (current_date - INTERVAL '10 years')
GROUP BY
    City, State
HAVING
    COUNT(*) > 0
ORDER BY
    num_severe_days DESC;
```

Demonstration Senario

All the features mentioned above will be available during demonstration. We might add some new features if time permits.

Notes:
1. Will make changes to the UI after the majors.
2. We don't have a lot of update/inserts/deletion queries as they don't make much sense since this is visualization project. That being said, inserts are done in a couple of transactions like signing up for weather alerts, daily weather forecasts and for logging.
3. Will add more database features later if they make sense.
4. For triggers, we've implemented one trigger for each valid use case that we can think of. No unnecessary stuff or features are implemented.
5. We've used SQLAlchemy for interaction with the database. Since it is ORM, the queries that we provided in the milestone 2 are not run as it is, like in the case of psycopg2.
6. For the most part, the queries are simple. But they will change a bit based on the selections that the user makes.

Github Repository Link: [SMV](#)