Step 1: Set Up Your Project Structure

Create Project Directory:

Create a new directory for your project, e.g., `weather-api`.

Create Subdirectories:

Inside the `weather-api` directory, create the following folders:

- `lambda`: For your Lambda function classes.
- `model`: For your data model classes.
- `utils`: For utility classes.

Create Java Classes:

- Inside the `lambda` folder, create:
  - `GetItemFunction.java`
  - `PostItemFunction.java`
- Inside the `model` folder, create:
  - `WeatherData.java`
- Inside the `utils` folder, create:
  - `DependencyFactory.java`

Step 2: Implement the Java Classes

WeatherData.java (Model):

- Define the data structure for weather data.
- Include fields for city name, weather ID, main weather type, description, and icon.
- Implement constructors and getter/setter methods.

DependencyFactory.java (Utils):

- Create methods to initialize the DynamoDB Enhanced Client and retrieve the table name from environment variables.

PostItemFunction.java (Lambda):

- Implement the `RequestHandler` interface.
- In the constructor, initialize the DynamoDB client, table name, and table schema.
- In `handleRequest`, extract the city name from the request parameters, fetch weather data from the OpenWeather API, and save it to DynamoDB.
- Handle exceptions and return appropriate HTTP responses.

GetItemFunction.java (Lambda):

- Implement the `RequestHandler` interface similarly.
- In the constructor, initialize the DynamoDB client and table schema.

- In `handleRequest`, scan the DynamoDB table for all weather data and return it as a JSON response.
- Handle exceptions and log errors appropriately.

Step 3: Build Your Project

Install Dependencies:

Ensure you have Maven or your chosen build tool set up to manage dependencies defined in `pom.xml`.

Compile the Code:

Run the build command (e.g., `mvn clean package` if using Maven) to compile your Java code and package it into a JAR file.

Step 4: Set Up AWS and Serverless Framework

Install the AWS CLI:

Install the AWS CLI and configure it using your AWS credentials:

```
aws configure
```

Install the Serverless Framework:

Install the Serverless Framework using Node.js:

```
npm install -g serverless
```

Step 5: Deploy to AWS

Deploy Your Application:

In the root of your project, run:

```
serverless deploy
```

This command packages your application, creates necessary resources (like API Gateway and DynamoDB), and deploys your Lambda functions.

Record the API Gateway Endpoint:

After deployment, take note of the API endpoint URLs for testing your application.

Step 6: Test the API

Test the POST Endpoint:

Use a tool like Postman or curl to send a POST request to add weather data:

```
curl -X POST 'https://<api-id>.execute-api.us-east-1.amazonaws.com/dev/weather?city=<CityName>'
```

Verify that the weather data is stored in DynamoDB.

Test the GET Endpoint:

Send a GET request to retrieve all weather data:

```
curl -X GET 'https://<api-id>.execute-api.us-east-1.amazonaws.com/dev/weather/all'
```

Check that it returns the correct weather data in JSON format.

Step 7: Set Up CI/CD with GitHub Actions

Create a GitHub Actions Workflow:

In your GitHub repository, create the directory `.github/workflows`.

Add a CI/CD YAML File:

Inside the `workflows` directory, create a `ci-cd.yml` file. Define the workflow to build the Java project and deploy it to AWS whenever changes are pushed to the main branch.

Configure GitHub Secrets:

In your repository settings, add secrets for `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` to enable GitHub Actions to deploy your application securely.

Summary

By following these steps, you'll have a fully functional AWS Weather App that utilizes Lambda functions to interact with DynamoDB and an external weather API. You can deploy and manage the app easily, and with the CI/CD pipeline, your application will automatically update on pushes to the main branch.