Java Weather App Deployment Using AWS and CI/CD with GitHub Actions

Aim

Implementing a Java application with the OpenWeather API and deploying it using AWS, along with CI/CD via GitHub Actions.

Introduction

This documentation outlines the process of creating a Java application that leverages the OpenWeather API to provide real-time weather information for cities. The application features two key API endpoints: a POST endpoint for submitting city names and storing their weather data in AWS DynamoDB, and a GET endpoint for retrieving weather information for all stored cities. The backend is hosted on AWS using Lambda functions, which enable serverless execution, and API Gateway to manage the API calls. Additionally, the project integrates a CI/CD pipeline using GitHub Actions to automate testing and deployment, ensuring seamless updates and scalability. This comprehensive guide will walk you through each step, from project setup to deployment, making it easier to build and maintain a robust weather application on AWS.

Comprehensive Overview

1. Project Description

This Java application provides weather information for cities using the OpenWeather API. It features two main API endpoints:

- **POST /weather:** Accepts a city name as a request parameter and stores its weather information in a DynamoDB table.
- **GET /weather:** Returns the weather information for all stored cities.

The application is deployed on AWS using Lambda for serverless computing and API Gateway to manage the API endpoints. Continuous Integration and Continuous Deployment (CI/CD) is implemented using GitHub Actions.

2. Technologies Used

- **Java:** The programming language used for developing the application.
- **AWS Lambda:** A serverless compute service that runs the application without provisioning servers.
- **AWS API Gateway:** Manages the API endpoints.
- **AWS DynamoDB:** A NoSQL database for storing weather information.
- **OpenWeather API:** Provides weather data for cities.

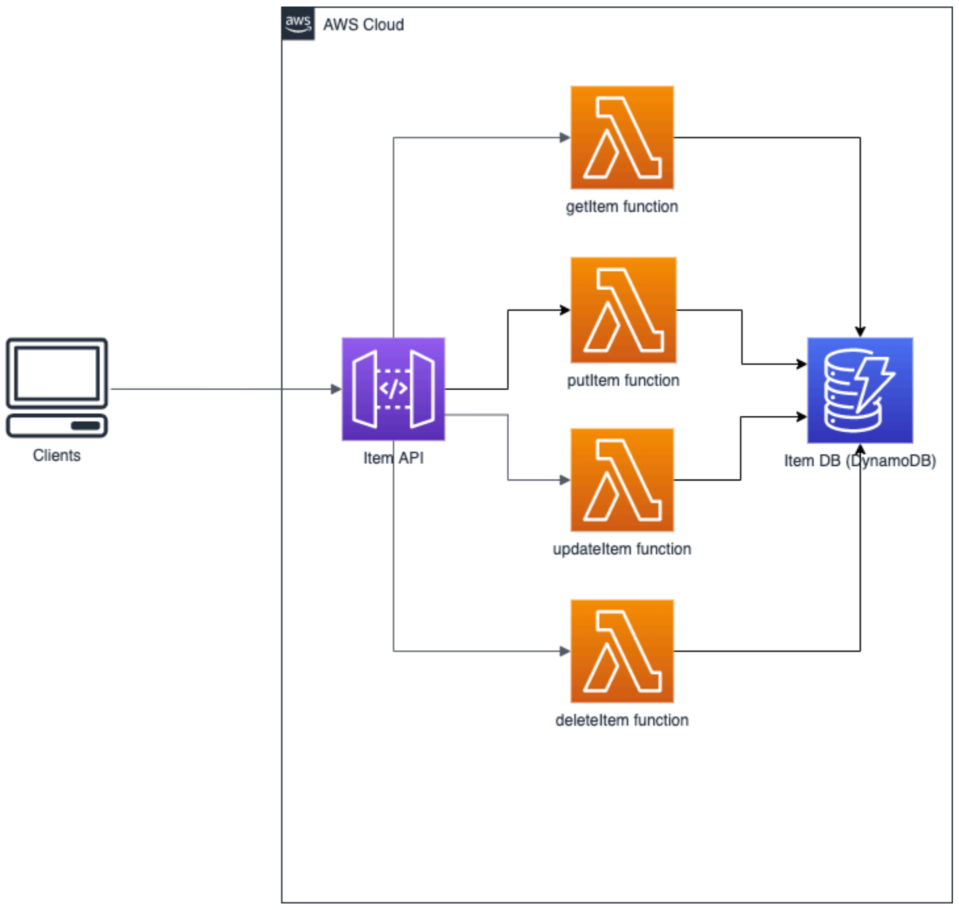
- **GitHub Actions:** Automates the CI/CD pipeline.

3. Prerequisites

Before starting, ensure you have the following:

- **Java Development Kit (JDK):** Version 11 or higher installed.
- **AWS Account:** To access AWS services.
- **IDE:** Such as IntelliJ IDEA or Eclipse for Java development.
- **Git:** Installed for version control.
- **Basic Knowledge:** Familiarity with Java, RESTful APIs, and AWS services.

Model Architecture Overview



The architecture of the application follows a structured approach, where different components interact seamlessly to perform operations related to items. Here's a breakdown of the architecture:

Client

The client is the front-end interface that users interact with. This could be a web application, mobile app, or any HTTP client. It sends HTTP requests to the API to perform operations such as creating, reading, updating, and deleting items.

Item API

The Item API serves as the intermediary between the client and the data layer (DynamoDB). It exposes a set of endpoints that the client can call:

- **POST /item**: For creating new items.
- **GET /item/{id}**: For retrieving a specific item.
- **PUT /item/{id}**: For updating an existing item.
- **DELETE /item/{id}**: For deleting an item.

The API receives requests from the client, processes them, and then delegates specific operations to the corresponding functions.

Function Handlers

The Item API connects to several function handlers that encapsulate the logic for interacting with DynamoDB:

- **getItem**: Retrieves an item from DynamoDB based on its unique identifier. It takes the item ID as a parameter and queries the database to return the item details.
- **putItem**: Adds a new item to DynamoDB. This function accepts item data from the API, constructs a DynamoDB item, and inserts it into the database.
- **updateItem**: Updates an existing item in DynamoDB. This function retrieves the item using its ID, modifies the necessary fields, and saves the updated item back to the database.
- **deleteItem**: Removes an item from DynamoDB. It takes the item ID as input and deletes the corresponding record from the database.

Item Database (DynamoDB)

DynamoDB is the NoSQL database used to store item data. It provides fast and scalable data storage, allowing the application to handle various operations efficiently. The data model typically includes attributes relevant to the items, such as item ID, name, description, and any other necessary fields.

Interaction Flow

1. **Client Request**: The client initiates an action by sending an HTTP request to the Item API.
2. **API Processing**: The Item API receives the request, determines the appropriate function to call (e.g., getItem, putItem, updateItem, or deleteItem), and forwards the necessary parameters.
3. **Database Interaction**: The function handler communicates with DynamoDB to perform the requested operation.
4. **Response**: After the operation is completed, the function returns the result (success message, item data, etc.) back to the API, which then sends the response to the client.

Implementation Steps

A. **Create a Java Application**

1. **Set Up Your Development Environment**:
   - Install the **Java Development Kit (JDK)** (version 17 or higher).
   - Choose an **IDE** (e.g., IntelliJ IDEA, Eclipse).
   - Ensure **Maven** s installed for dependency management.
2. **Create a New Java Project**:
   - Create a new project in your IDE.
   - generate a pom.xml
3. **Add Dependencies**:

Include necessary dependencies for AWS SDK and any additional libraries (e.g., for JSON processing):

```
○ <project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
○   <modelVersion>4.0.0</modelVersion>
○
○   <groupId>com.home.amazon.serverless</groupId>
○   <artifactId>aws-serverless-framework-app</artifactId>
○   <packaging>jar</packaging>
○   <version>dev</version>
○   <name>aws-serverless-framework-app</name>
○
○   <properties>
○     <maven.compiler.source>11</maven.compiler.source>
○     <maven.compiler.target>11</maven.compiler.target>
○     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
○     <jackson.version>2.12.4</jackson.version>
○     <log4j.version>2.14.1</log4j.version>
○     <aws.java.sdk.version>2.15.79</aws.java.sdk.version>
○     <aws.lambda.java.version>1.2.1</aws.lambda.java.version>
○     <junit5.version>5.7.2</junit5.version>
○     <mockito.version>3.12.1</mockito.version>
○   </properties>
○
○   <dependencyManagement>
○     <dependencies>
○       <dependency>
○         <groupId>software.amazon.awssdk</groupId>
○         <artifactId>bom</artifactId>
○         <version>${aws.java.sdk.version}</version>
○         <type>pom</type>
○         <scope>import</scope>
○       </dependency>
○     </dependencies>
```

```
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-log4j2</artifactId>
    <version>1.2.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>${log4j.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>${log4j.version}</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>${jackson.version}</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>${jackson.version}</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>${jackson.version}</version>
  </dependency>

  <dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>dynamodb-enhanced</artifactId>
    <exclusions>
      <exclusion>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>netty-nio-client</artifactId>
      </exclusion>
      <exclusion>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>apache-client</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-core</artifactId>
    <version>${aws.lambda.java.version}</version>
  </dependency>

  <dependency>
```

```
○       <groupId>software.amazon.awssdk</groupId>
○       <artifactId>url-connection-client</artifactId>
○     </dependency>
○
○     <dependency>
○       <groupId>com.amazonaws</groupId>
○       <artifactId>aws-lambda-java-events</artifactId>
○       <version>3.9.0</version>
○     </dependency>
○
○     <!-- Test Dependencies -->
○     <dependency>
○       <groupId>org.junit.jupiter</groupId>
○       <artifactId>junit-jupiter</artifactId>
○       <version>${junit5.version}</version>
○       <scope>test</scope>
○     </dependency>
○     <dependency>
○       <groupId>org.mockito</groupId>
○       <artifactId>mockito-junit-jupiter</artifactId>
○       <version>${mockito.version}</version>
○       <scope>test</scope>
○     </dependency>
○     <dependency>
○       <groupId>org.mockito</groupId>
○       <artifactId>mockito-inline</artifactId>
○       <version>${mockito.version}</version>
○       <scope>test</scope>
○     </dependency>
○
○   </dependencies>
○
○   <build>
○     <plugins>
○       <!--
○         Using the Apache Maven Shade plugin to package the jar
○
○         "This plugin provides the capability to package the artifact
○         in an uber-jar, including its dependencies and to shade - i.e. rename -
○         the packages of some of the dependencies."
○
○         Link: https://maven.apache.org/plugins/maven-shade-plugin/
○       -->
○       <plugin>
○         <groupId>org.apache.maven.plugins</groupId>
○         <artifactId>maven-surefire-plugin</artifactId>
○         <version>2.22.2</version>
○       </plugin>
○       <plugin>
○         <groupId>org.apache.maven.plugins</groupId>
○         <artifactId>maven-shade-plugin</artifactId>
○         <version>2.3</version>
○         <configuration>
○           <createDependencyReducedPom>false</createDependencyReducedPom>
○         </configuration>
○         <executions>
○           <execution>
○             <phase>package</phase>
○             <goals>
○               <goal>shade</goal>
○             </goals>
○             <configuration>
○               <transformers>
○                 <transformer
○
  implementation="com.github.edwgiz.mavenShadePlugin.log4j2CacheTransformer.PluginsCa
  cheFileTransformer">
○                 </transformer>
○               </transformers>
○             </configuration>
○           </execution>
○         </executions>
○         <dependencies>
○           <dependency>
○             <groupId>com.github.edwgiz</groupId>
○             <artifactId>maven-shade-plugin.log4j2-cachefile-
  transformer</artifactId>
○             <version>2.8.1</version>
○           </dependency>
○         </dependencies>
○       </plugin>
○       <plugin>
○         <groupId>org.apache.maven.plugins</groupId>
○         <artifactId>maven-compiler-plugin</artifactId>
○         <configuration>
○           <source>11</source>
○           <target>11</target>
○         </configuration>
○       </plugin>
○     </plugins>
○   </build>
○
○ </project>
```

4. Implement the Model and Services

a. Create the Item Model:

```
○ package com.home.amazon.serverless.model;
○ import com.fasterxml.jackson.annotation.JsonProperty;
○ import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.DynamoDbBean;
○ import
  software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.DynamoDbPartitionKey;
○
○ import java.util.Objects;
○
○ @DynamoDbBean
○ public class WeatherData {
○
○     private String cityName;
○     private int id;
○     private String main;
```

```
    private String description;
    private String icon;

    public WeatherData() {}

    public WeatherData(String cityName, int id, String main, String description, String icon) {
        this.cityName = cityName;
        this.id = id;
        this.main = main;
        this.description = description;
        this.icon = icon;
    }

    @DynamoDbPartitionKey
    public String getCityName() {
        return cityName;
    }

    public void setCityName(String cityName) {
        this.cityName = cityName;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getMain() {
        return main;
    }

    public void setMain(String main) {
        this.main = main;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getIcon() {
        return icon;
    }

    public void setIcon(String icon) {
        this.icon = icon;
    }
}
```

b. Implement DependencyFactory for DynamoDB :

- The **DependencyFactory** class is a utility designed to create an instance of **DynamoDbEnhancedClient** for interacting with AWS DynamoDB. It utilizes environment variables to configure the client, fetching AWS credentials and the specified region from the environment. The **dynamoDbEnhancedClient** method constructs and returns the enhanced client, which simplifies operations with DynamoDB, while the **tableName** method retrieves the name of the DynamoDB table from another environment variable. This design promotes clean code practices by encapsulating the client configuration logic, ensuring that other parts of the application can easily obtain a ready-to-use client without dealing with the underlying complexity.

```
package com.home.amazon.serverless.utils;

import software.amazon.awssdk.auth.credentials.EnvironmentVariableCredentialsProvider;
import software.amazon.awssdk.core.SdkSystemSetting;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.http.urlconnection.UrlConnectionHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

public class DependencyFactory {

    public static final String ENV_VARIABLE_TABLE = "TABLE";

    private DependencyFactory() {
    }

    /**
     * @return an instance of DynamoDbClient
     */

    public static DynamoDbEnhancedClient dynamoDbEnhancedClient() {
        return DynamoDbEnhancedClient.builder()
                .dynamoDbClient(DynamoDbClient.builder()

.credentialsProvider(EnvironmentVariableCredentialsProvider.create())

.region(Region.of(System.getenv(SdkSystemSetting.AWS_REGION.environmentVariable())))
                        .httpClientBuilder(UrlConnectionHttpClient.builder())
                        .build())
                .build();
    }

    public static String tableName() {
        return System.getenv(ENV_VARIABLE_TABLE);
    }
}
```

c. Define the GetItemFucntion for lamda:

- The **GetItemFunction** class is an AWS Lambda function that handles requests to retrieve weather data stored in DynamoDB. It implements the **RequestHandler** interface, allowing it to process API Gateway events. Upon initialization, the class creates an instance of **DynamoDbEnhancedClient** and retrieves the table name from environment variables.

- In the **handleRequest** method, it scans the DynamoDB table for all **WeatherData** entries and converts them to JSON format using Jackson's **ObjectMapper**. If the conversion is successful, it returns a 200 HTTP status code along with the weather data in the response body. In case of a JSON processing error, it logs the error and returns a 500 status code with an error message.
- This structure provides a clean way to access and return weather information through an API endpoint.

```java
package com.home.amazon.serverless.lambda;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.home.amazon.serverless.model.WeatherData;
import com.home.amazon.serverless.utils.DependencyFactory;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;

import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

public class GetItemFunction implements RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {
    private final DynamoDbEnhancedClient dbClient;
    private final String tableName;
    private final TableSchema<WeatherData> weatherTableSchema;

    public GetItemFunction() {
        dbClient = DependencyFactory.dynamoDbEnhancedClient();
        tableName = DependencyFactory.tableName();
        weatherTableSchema = TableSchema.fromBean(WeatherData.class);
    }

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent input, Context context) {
        String response = "";
        LambdaLogger logger = context.getLogger();
        DynamoDbTable<WeatherData> weatherTable = dbClient.table(tableName, weatherTableSchema);

        try {
            List<WeatherData> allWeatherData = weatherTable.scan().items().stream().collect(Collectors.toList());
            response = new ObjectMapper().writeValueAsString(allWeatherData);
        } catch (JsonProcessingException e) {
            logger.log("Failed to create a JSON response: " + e);
            return new APIGatewayProxyResponseEvent()
                    .withStatusCode(500)
                    .withBody("Error processing weather data");
        }

        return new APIGatewayProxyResponseEvent()
                .withStatusCode(200)
                .withIsBase64Encoded(Boolean.FALSE)
                .withHeaders(Collections.singletonMap("Content-Type",
"application/json"))
                .withBody(response);
    }


}
```

d. Define the PostItemFucntion for lambda:

- The **PostItemFunction** class is a critical component of the serverless application that enables users to store and manage weather data in a DynamoDB table based on city names provided through API requests. By implementing the **RequestHandler** interface, this AWS Lambda function handles incoming requests from API Gateway, processes the specified city name, and interacts with the OpenWeather API to retrieve real-time weather information.
- The extracted data is encapsulated in a **WeatherData** object, which is then updated in DynamoDB. This function is essential for ensuring that the application can dynamically respond to user queries about weather conditions, facilitating efficient data storage and retrieval. Its robust error handling improves user experience by providing clear feedback on the success or failure of requests, while integration with AWS services ensures scalability and reliability in a serverless environment.

```
package com.home.amazon.serverless.lambda;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.JsonNode;

import com.home.amazon.serverless.model.WeatherData;
import com.home.amazon.serverless.utils.DependencyFactory;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.Collections;
```

```java

public class PostItemFunction implements RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    private final DynamoDbEnhancedClient dbClient;
    private final String tableName;
    private final TableSchema<WeatherData> weatherTableSchema;
    private final String API_KEY = "3301519246e749b9b45247cd0fb0291c";
    private final String API_URL = "https://api.openweathermap.org/data/2.5/weather?q=%s&appid=" + API_KEY;

    public PostItemFunction() {
        dbClient = DependencyFactory.dynamoDbEnhancedClient();
        tableName = DependencyFactory.tableName();
        weatherTableSchema = TableSchema.fromBean(WeatherData.class);
    }

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent request, Context context) {
        String cityName = request.getQueryStringParameters().get("city");
        String responseBody = "";

        if (cityName != null && !cityName.isEmpty()) {
            try {
                String weatherData = fetchWeatherData(cityName);
                WeatherData weatherItem = extractWeatherData(weatherData, cityName);

                if (weatherItem != null) {
                    DynamoDbTable<WeatherData> weatherTable = dbClient.table(tableName, weatherTableSchema);
                    WeatherData savedItem = weatherTable.updateItem(weatherItem);
                    responseBody = new ObjectMapper().writeValueAsString(savedItem);
                }
            } catch (Exception e) {
                context.getLogger().log("Error processing request: " + e.getMessage());
                return new APIGatewayProxyResponseEvent()
                        .withStatusCode(500)
                        .withBody("Error processing request");
            }
        } else {
            return new APIGatewayProxyResponseEvent()
                    .withStatusCode(400)
                    .withBody("City name is required");
        }

        return new APIGatewayProxyResponseEvent()
                .withStatusCode(200)
                .withIsBase64Encoded(Boolean.FALSE)
                .withHeaders(Collections.emptyMap())
                .withBody(responseBody);
    }

    private String fetchWeatherData(String cityName) throws Exception {
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(String.format(API_URL, cityName)))
                .build();

        HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
        return response.body();
    }

    private WeatherData extractWeatherData(String jsonData, String cityName) throws
JsonProcessingException {
        ObjectMapper objectMapper = new ObjectMapper();
        JsonNode rootNode = objectMapper.readTree(jsonData);
        JsonNode weatherNode = rootNode.path("weather").get(0);

        return new WeatherData(
                cityName,
                weatherNode.path("id").asInt(),
                weatherNode.path("main").asText(),
                weatherNode.path("description").asText(),
                weatherNode.path("icon").asText()
        );
    }
}
```

5. **Logging using Log4j2**

- The provided XML configuration sets up logging for an AWS Lambda function using Log4j2, ensuring effective monitoring and troubleshooting. It specifies a Lambda appender that directs log messages to the AWS logging service, with a detailed pattern that includes the timestamp, AWS request ID, log level, logger name, line number, and the actual log message.
- This structured format enhances the visibility of log entries, making it easier to trace specific requests and diagnose issues. By configuring the root logger at the DEBUG level, the setup captures all relevant log messages, which is essential for maintaining high-quality serverless applications.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration packages="com.amazonaws.services.lambda.runtime.log4j2">
  <Appenders>
    <Lambda name="Lambda">
      <PatternLayout>
        <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1}:%L -
%m%n</pattern>
      </PatternLayout>
    </Lambda>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="Lambda" />
    </Root>
  </Loggers>
</Configuration>
```

6. Define YAML configuration for AWS serverless framework

```yaml
service: Weather-api
provider:
  name: aws
  runtime: java11
  environment:
    TABLE: ${self:service}-${sls:stage}-WeatherDataTable-${sls:instanceId}
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:DescribeTable
        - dynamodb:Query
        - dynamodb:Scan
        - dynamodb:GetItem
        - dynamodb:PutItem
        - dynamodb:UpdateItem
        - dynamodb:DeleteItem
      Resource: "arn:aws:dynamodb:${aws:region}:${aws:accountId}:table/${self:provider.environment.TABLE}"
  stage: dev
  region: us-east-1

package:
  artifact: target/aws-serverless-framework-app-dev.jar

functions:
  getWeatherData:
    handler: com.home.amazon.serverless.lambda.PostItemFunction
    events:
      - http:
          path: /weather
          method: post
          request:
            parameters:
              querystrings:
                city: true
  getAllWeatherData:
    handler: com.home.amazon.serverless.lambda.GetItemFunction
    events:
      - http:
          path: /weather/all
          method: get

resources:
  Resources:
    WeatherDataDynamoDbTable:
      Type: 'AWS::DynamoDB::Table'
      Properties:
        AttributeDefinitions:
          - AttributeName: cityName
            AttributeType: S
        KeySchema:
          - AttributeName: cityName
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
        SSESpecification:
          SSEEnabled: true
        TableName: ${self:provider.environment.TABLE}
```

7. **CloudFormation Implementation :**

- CloudFormation is a service provided by Amazon Web Services (AWS) that allows you to define and provision AWS infrastructure using code. With CloudFormation, you create a template in JSON or YAML format that describes the resources you need, such as EC2 instances, S3 buckets, and VPCs.
- This template acts as a blueprint for deploying your infrastructure consistently and repeatedly. You can version control your templates, making it easier to manage changes and replicate environments. Additionally, CloudFormation manages dependencies between resources, ensuring they are created in the correct order. Overall, it helps automate and simplify the process of setting up and managing AWS resources.

Using AWS CloudFormation in a project involves several steps:

1. **Define Your Infrastructure**: Start by identifying the AWS resources your project needs (like EC2 instances, RDS databases, etc.).
2. **Create a CloudFormation Template**: Write a template in JSON or YAML format that describes your desired resources and their configurations. This template includes parameters, resources, outputs, and possibly conditions.
3. **Validate the Template**: Use the AWS CloudFormation console or CLI to validate your template. This helps catch any syntax errors or issues before deployment.
4. **Deploy the Stack**: Use the CloudFormation console, AWS CLI, or SDKs to create a stack from your template. A stack is a collection of AWS resources that you can manage as a single unit.
5. **Monitor the Deployment**: After creating the stack, monitor its progress in the CloudFormation console. You can check for errors or view the status of resource creation.
6. **Update the Stack**: If you need to make changes, modify your template and update the stack. CloudFormation handles the changes while keeping the existing resources intact.
7. **Delete the Stack**: When the project is complete or if you need to tear down the resources, you can delete the stack, and CloudFormation will automatically clean up all associated resources.
8. **Version Control**: Store your CloudFormation templates in a version control system (like Git) to track changes and collaborate with your team.

This approach allows for consistent and repeatable infrastructure provisioning, making it easier to manage environments throughout your project lifecycle.

B. **Set Up DynamoDB**

- Go to the AWS Management Console and create a DynamoDB table (e.g., WeatherData).
- Define a primary key (e.g., requestId).

C. **Deploy the Lambda Function**

- In the AWS Lambda console, create a new Lambda function.
- Upload your packaged JAR file.
- Set up the execution role with permissions to access DynamoDB.

D. **Set Up API Gateway**

1. **Create a New API**:
   - In API Gateway, create a new REST API.
2. **Create Resource and Methods**:
   - Create a resource (e.g., /weather).
   - Create a GET method and integrate it with your Lambda function.
3. **Deploy the API**:
   - Create a new stage (e.g., dev) and deploy the API.

E. **Test Your API**

- Use Postman or curl to test your API endpoint (e.g., https://your-api-id.execute-api.region.amazonaws.com/dev/weather?city=London).
- Check if it retrieves weather data from OpenWeather.

F. **CI/CD Pipeline using Github Actions**

Implement a CI/CD pipeline using GitHub Actions for your Java application deployed on AWS Lambda, API Gateway, and DynamoDB which is a great way to automate the deployment process.

1. **Prepare Your GitHub Repository**

- Ensure your Java project is in a GitHub repository. Include your pom.xml (for Maven) to manage dependencies.

2. **Create a GitHub Actions Workflow**

- Create a directory for GitHub Actions workflows in your repository:
- `mkdir -p .github/workflows`
- Then create a YAML file for your workflow, e.g., ci-cd-pipeline.yml.

3. **Define the Workflow**

```
name: Deploy Serverless Application

on:
  push:
    branches:
      - main  # or your default branch name

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2

    - name: Set up JDK 11
      uses: actions/setup-java@v2
      with:
        java-version: '11'
        distribution: 'adopt'

    - name: Cache Maven packages
      uses: actions/cache@v2
      with:
        path: ~/.m2
        key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
        restore-keys: ${{ runner.os }}-m2

    - name: Build with Maven
      run: mvn clean package

    - name: Set up Node.js
      uses: actions/setup-node@v2
      with:
        node-version: '18'  # Update to Node.js version 18 as required

    - name: Install Serverless Framework
      run: npm install -g serverless

    - name: Deploy to AWS
      env:

        SERVERLESS_ACCESS_KEY: AK0O7eNtGPNrGn29QOoROH6eaKy5vzDEXEHX8zuHmXsdl  # Your Serverless Access Key if needed
      run: |
        serverless deploy --stage dev
```

4. **Set Up Secrets in GitHub**

To securely store your AWS credentials:

1. Go to your GitHub repository.
2. Navigate to **Settings** > **Secrets and variables** > **Actions**.
3. Add the following secrets:
   - AWS_ACCESS_KEY_ID: Your AWS access key ID.
   - AWS_SECRET_ACCESS_KEY: Your AWS secret access key.

5. **Configure Your Lambda Function**

Ensure your Lambda function is set to use the correct handler. The handler should match the package structure and class name in your JAR.

6. **Test the Workflow**

1. Make a change to your Java code or any other files.
2. Push your changes to the main branch (or your specified branch).
3. Go to the **Actions** tab in your GitHub repository to see the workflow run.