**Title:**

**PoC – Migrating AWS Lambda Function from Python to Java for DynamoDB**

**1. Objective**

To evaluate the feasibility and performance of converting an existing AWS Lambda function written in Python into Java. The function queries a DynamoDB table using the PoC aims to validate Java's compatibility with AWS Lambda for this task, while comparing size and execution behavior with the Python version.

**2. Problem Statement**

The current Lambda function implemented in Python uses boto3 to query a DynamoDB table via a GSI and return filtered, sorted data. To align with organizational standards favoring Java-based systems, the function needs to be converted to Java. However, AWS Lambda does not offer an inline Java editor, which introduces deployment and size considerations.

**3. Workflow**

**A diagram of a computer program

AI-generated content may be incorrect.**

**4. Scope**

* + Porting existing Python code to Java
  + Deploying Java Lambda using S3
  + Querying DynamoDB using GSI in Java
  + Comparison of file size and performance
  + Returning JSON-formatted data

**5. Tools & Technologies Used**

|  |  |
| --- | --- |
| * Component | * Technology |
| * Original Code | * Python (boto3) |
| * Converted Code | * Java (AWS SDK v2) |
| * Cloud Service | * AWS Lambda |
| * Database | * Amazon DynamoDB |
| * Deployment | * AWS S3 (for Java JAR) |
| * JSON Parser | * Jackson |

**6. Architecture / Flow**

[Client/Event Trigger]

↓

[AWS Lambda (Java, uploaded via S3)]

↓

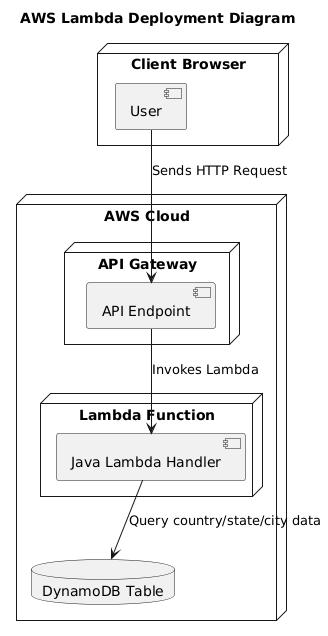
[Query DynamoDB (GSI)]

↓

[Process, Filter, Sort Data]

↓

[Return JSON Response]



A close-up of a list

AI-generated content may be incorrect.

**7. Implementation Steps**

**Review Python Lambda code**

* Understand the logic for GSI query, filtering, and sorting.

**Set up Java project (Maven)**

* Add AWS SDK v2 and Jackson dependencies.

**Develop Java Lambda handler**

* Implement RequestHandler interface.
* Define input/output models.

**Query DynamoDB with GSI in Java**

* Use DynamoDbClient, QueryRequest, and ExpressionAttributeValues.

**Process and sort the results**

* Use Java collections and stream API for sorting/filtering.

**Build JAR file (18MB)**

* Due to the size exceeding 10MB, the JAR was uploaded to **S3**.

**Deploy Lambda using S3 reference**

* Configure Lambda to load the handler from the uploaded JAR in S3.

**Compare performance and size**

* File size and runtime behavior were compared with Python.

**8. Assumptions & Limitations**

* Lambda does not support inline editing for Java.
* Java JAR file exceeded the 10MB limit, hence required S3 upload.
* Sorting is handled client-side (within Lambda), not in DynamoDB.
* Cold start time for Java Lambdas may affect performance.

**3. DynamoDB Table Design**

Table Name: From environment variable DbTable

Schema:

* PK – Primary Partition Key
* PK1 / SK1 – Keys for Global Secondary Index (GSI1)

Example Structure:

* PK = CountryList# → stores a list of countries
* PK1 = Country#India, SK1 = Country#Char → stores states of India
* PK1 = Country#India, SK1 = State#Char → stores cities of states within India

**4. API Behavior**

Query ParamsResponseDynamoDB QueryNoneList of countries

PK = CountryList#Country=IndiaList of statesPK1 = Country#India AND

SK1 = Country#CharCountry=India&State=KarnatakaList of cities

PK1 = Country#India AND SK1 = State#Char

**5. Java Class Structure**

Class Name: LambdaHandler

Implements RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent>

**Main Components:**

**Initialization**

private final DynamoDbClient dynamoDbClient = DynamoDbClient.create();

private final String tableName = System.getenv("DbTable");

private final ObjectMapper objectMapper = new ObjectMapper();

Entry Method

public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent event, Context context)

**6. Code Logic Breakdown**

**Case 1: No Query Params (Get Countries)**

* Query PK = CountryList#
* Extract Countries attribute from response items
* Parse list of countries from a List<AttributeValue**>**

**Case 2: Only Country Param (Get States)**

* Query using GSI1 with PK1 = Country#<Country> and SK1 = Country#Char
* Extract States attribute from response as a Map<String, AttributeValue>
* Convert each state object to a simple map and sort by name

**Case 3: Country + State Params (Get Cities)**

* Query using GSI1 with PK1 = Country#<Country> and SK1 = State#Char
* Filter by State\_Name
* Extract Cities\_Name as List<AttributeValue> and convert to city list

**9. Environment and Setup**

**Java Version:** 17 (set in AWS Lambda console)

**Build Tool:** Maven

**Dependencies:**

software.amazon.awssdk:dynamodb

com.fasterxml.jackson.core:jackson-databind

**Sample pom.xml**

<dependencies>

  <dependency>

    <groupId>software.amazon.awssdk</groupId>

    <artifactId>dynamodb</artifactId>

  </dependency>

  <dependency>

    <groupId>com.fasterxml.jackson.core</groupId>

    <artifactId>jackson-databind</artifactId>

  </dependency>

</dependencies>

**9. Headers in Response**

{

  "Content-Type": "application/json",

  "Access-Control-Allow-Headers": "X-Api-Key, Content-Type, X-Amz-Date, Accept",

  "Access-Control-Allow-Origin": "\*",

  "Access-Control-Allow-Methods": "OPTIONS,GET"

}

**9. Results / Findings**

A screenshot of a computer

AI-generated content may be incorrect.

|  |  |  |
| --- | --- | --- |
| Metric | Python Lambda | Java Lambda |
| File Size | ~2 MB | ~18 MB |
| Performance | Fast | Slightly slower (due to cold start) |
| Deployment Ease | Easy (inline) | Requires S3 upload |
| Output Match | ✓ | ✓ |

* Java Lambda function successfully replicated the Python behavior.
* Output JSON matched in structure and content.
* Slight delay in Java cold start but acceptable for the use case.

**10. Conclusion**

The PoC confirms that the existing Python Lambda can be effectively converted to Java. While Java introduces deployment overhead and larger package size, it integrates well with AWS Lambda and meets the functional requirements.

**11. Recommendations / Next Steps**

* Optimize Java JAR size by excluding unnecessary dependencies.
* Consider using lighter SDK options for better performance.
* Add unit testing and logging for production readiness.
* Proceed with full-scale migration if Java standardization is preferred.