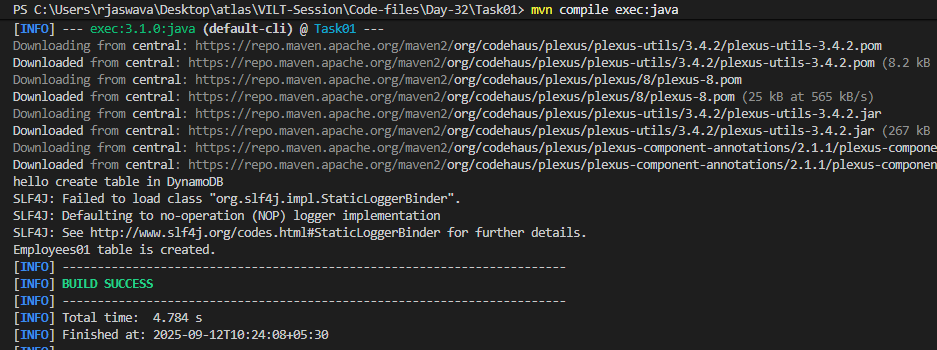
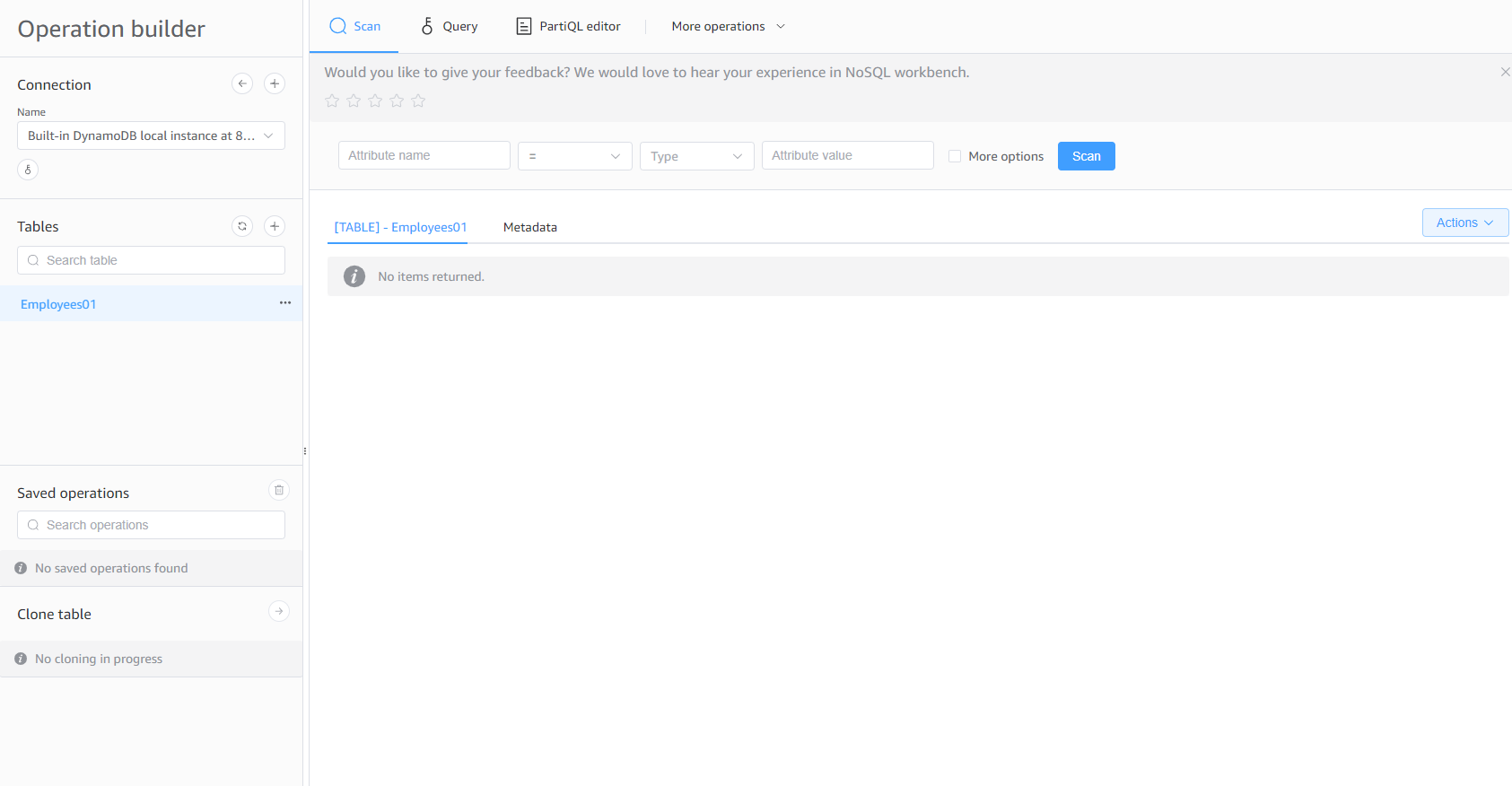
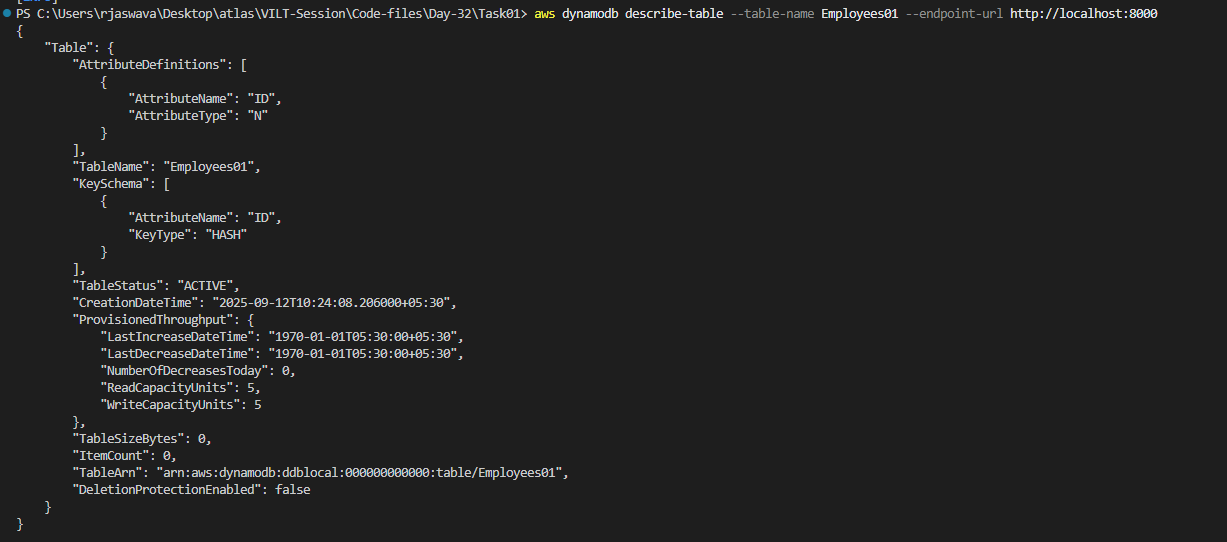
Task 01:

Create a table using Java code and check if the table is created.







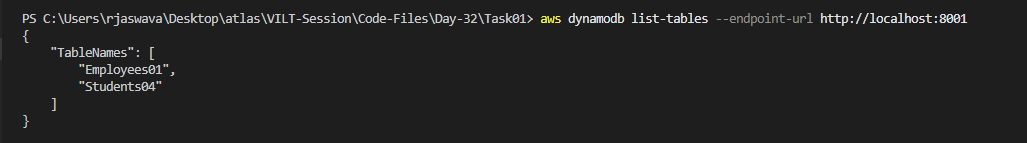
Task 02:

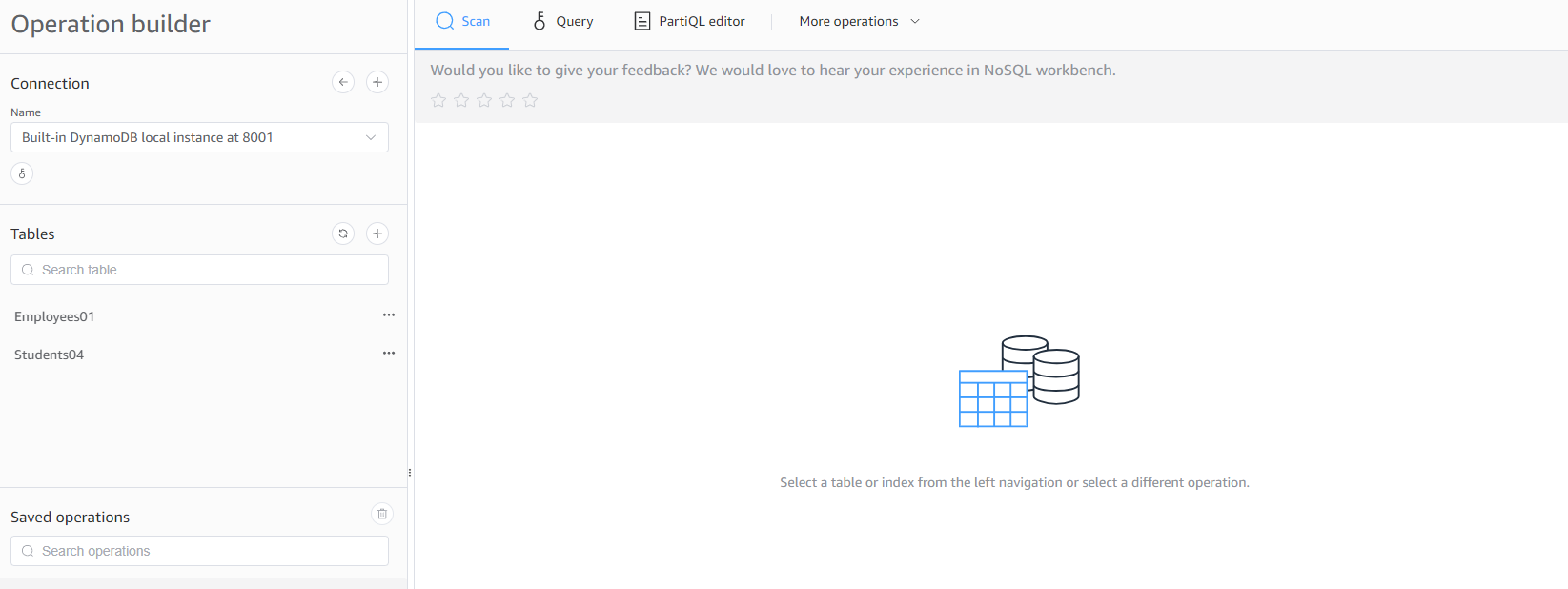
Using the same above java code

Change the post no and table name

To see the table reflecting in your cli prompt:

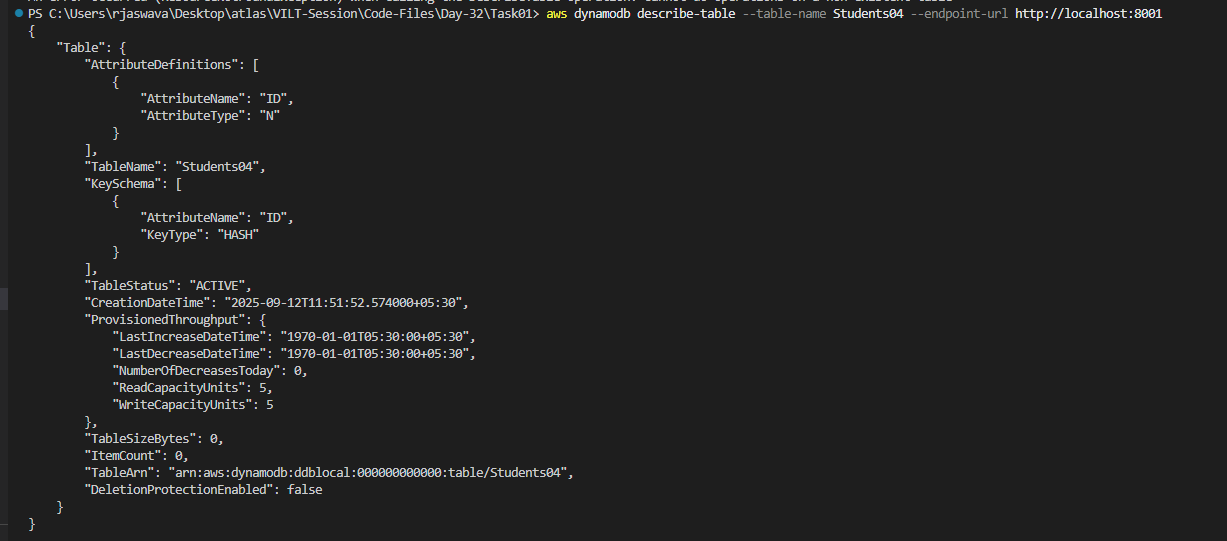
Note: your server should be running on the same post no:





Task 03:

For the same above tables see the description of the tables..



Task 04:

Loading / inserting data to the table

Plz make sure your json file is these in the resources folder

Output:

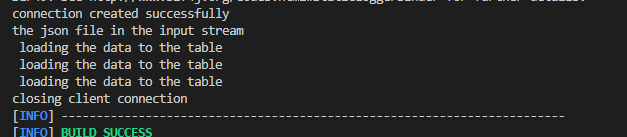
the json file in the input stream

 loading the data to the table

 loading the data to the table

 loading the data to the table

closing client connection

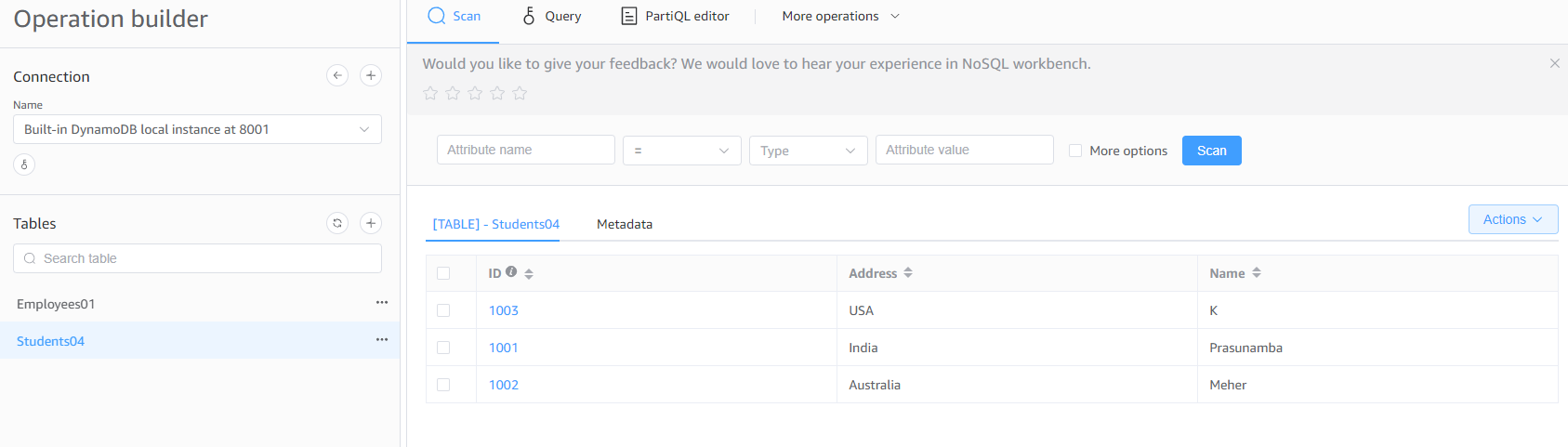


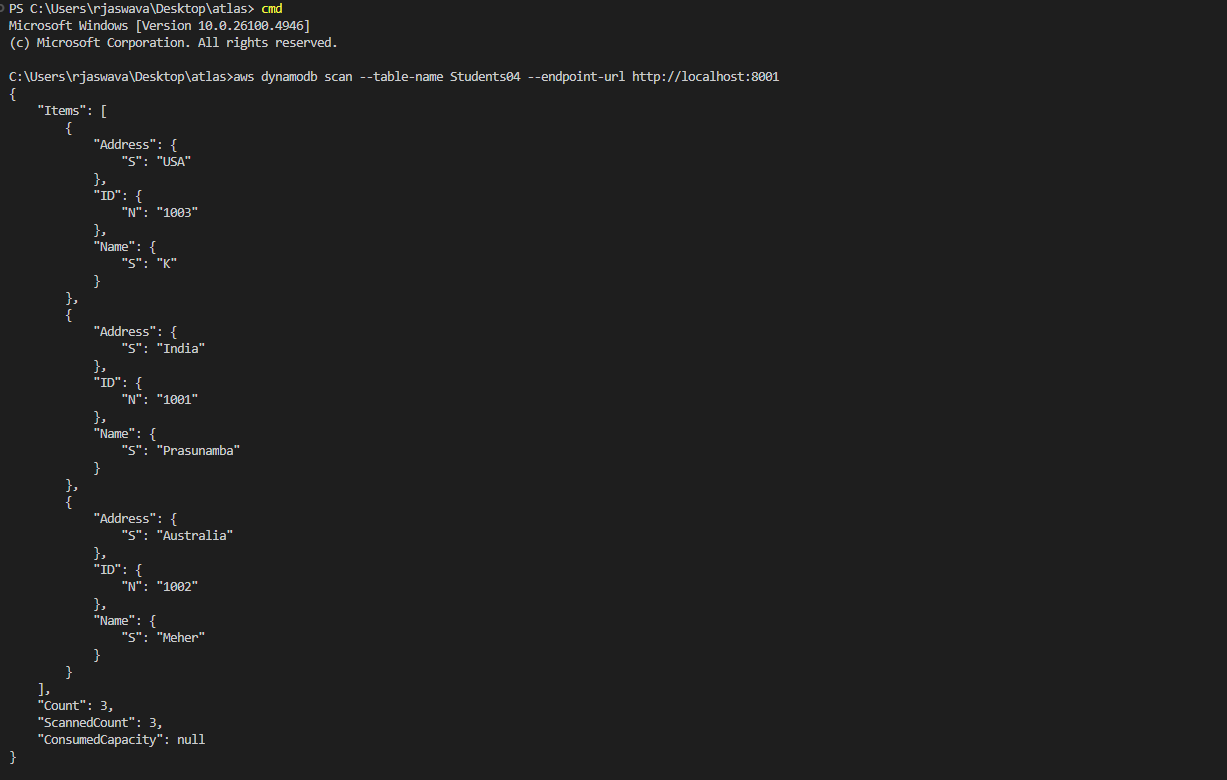
To check if the table is created… in the server

Your  server should be  open on the same port no

In another cmd promt type the below command to scan the items:

aws dynamodb scan --table-name Employees04 --endpoint-url http://localhost:8000





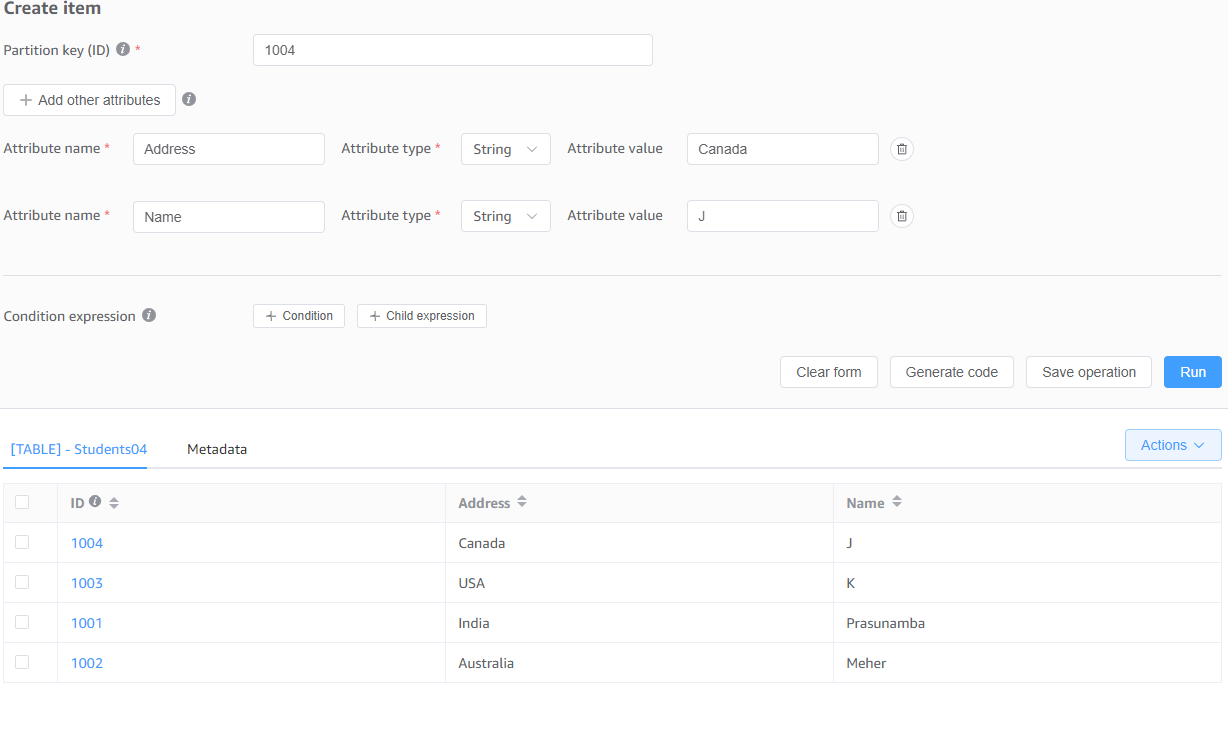
Task 05:

Reading / Scanning the data from the server.. Using java code..



Task 06:

Insert an item in the nosql workbench.. And check if the item reflects in your java output…





Task 07:

How do you lower the cost of DynomaDB..explain ways to do so…

Here are simple ways to lower DynamoDB costs:

1. Choose the right capacity mode:

- Use On-Demand mode if your traffic is unpredictable

- Use Provisioned mode if your traffic is steady and predictable

2. Optimize table design:

- Avoid storing unnecessary data

- Use shorter attribute names

- Compress large text or binary data before storing

3. Use TTL (Time To Live):

- Automatically delete old/expired items

- Helps reduce storage costs

- Good for temporary data like sessions or logs

4. Smart use of indexes:

- Only create indexes you really need

- Remove unused indexes

- Use sparse indexes when possible

5. Enable DynamoDB Auto Scaling:

- Automatically adjusts capacity based on actual usage

- Helps avoid over-provisioning

- Works with provisioned capacity mode

6. Use DAX (DynamoDB Accelerator):

- Caches frequently accessed data

- Reduces the number of read operations

- Can lower costs for read-heavy applications

7. Monitor and optimize:

- Use AWS CloudWatch to track usage

- Look for expensive operations

- Adjust capacity based on actual needs

8. Consider Reserved Capacity:

- Buy capacity in advance for heavy workloads

- Can save up to 72% compared to standard pricing

- Good for predictable, long-term usage

Task 08:

Update item details using java code.. And check if it reflects in the server..

package org.example;

import software.amazon.awssdk.auth.credentials.AwsBasicCredentials;

import software.amazon.awssdk.auth.credentials.StaticCredentialsProvider;

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

import software.amazon.awssdk.services.dynamodb.model.\*;

import java.net.URI;

import java.util.HashMap;

import java.util.Map;

public class UpdateItem {

    public static void main(String[] args) {

        AwsBasicCredentials awsCreds = AwsBasicCredentials.create("fakeaccess", "fakeaccess");

        DynamoDbClient client = DynamoDbClient.builder()

                .endpointOverride(URI.create("http://localhost:8001"))

                .region(Region.AP\_SOUTH\_1)

                .credentialsProvider(StaticCredentialsProvider.create(awsCreds))

                .build();

        String tableName = "Students04";

        try {

            Map<String, AttributeValue> key = new HashMap<>();

            key.put("ID", AttributeValue.builder().n("1001").build());

            Map<String, AttributeValueUpdate> updates = new HashMap<>();

            updates.put("Address", AttributeValueUpdate.builder()

                    .value(AttributeValue.builder().s("Delhi").build())

                    .action(AttributeAction.PUT)

                    .build());

            UpdateItemRequest request = UpdateItemRequest.builder()

                    .tableName(tableName)

                    .key(key)

                    .attributeUpdates(updates)

                    .returnValues(ReturnValue.ALL\_NEW)

                    .build();

            UpdateItemResponse response = client.updateItem(request);

            if (response.attributes() != null) {

                for (Map.Entry<String, AttributeValue> entry : response.attributes().entrySet()) {

                    System.out.println(entry.getKey() + ": " + entry.getValue());

                }

            }

        } catch (Exception e) {

            System.err.println("Error updating item: " + e.getMessage());

            e.printStackTrace();

        } finally {

            client.close();

        }

        try {

            ScanRequest scanRequest = ScanRequest.builder()

                    .tableName(tableName)

                    .build();

            ScanResponse scanResponse = client.scan(scanRequest);

            for (Map<String, AttributeValue> item : scanResponse.items()) {

                System.out.println("\nItem:");

                for (Map.Entry<String, AttributeValue> entry : item.entrySet()) {

                    System.out.println(entry.getKey() + ": " + entry.getValue());

                }

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}

Task 09:

Delete a particular item from the table …

 1002…

