Day 34

Employee ID: 201933938

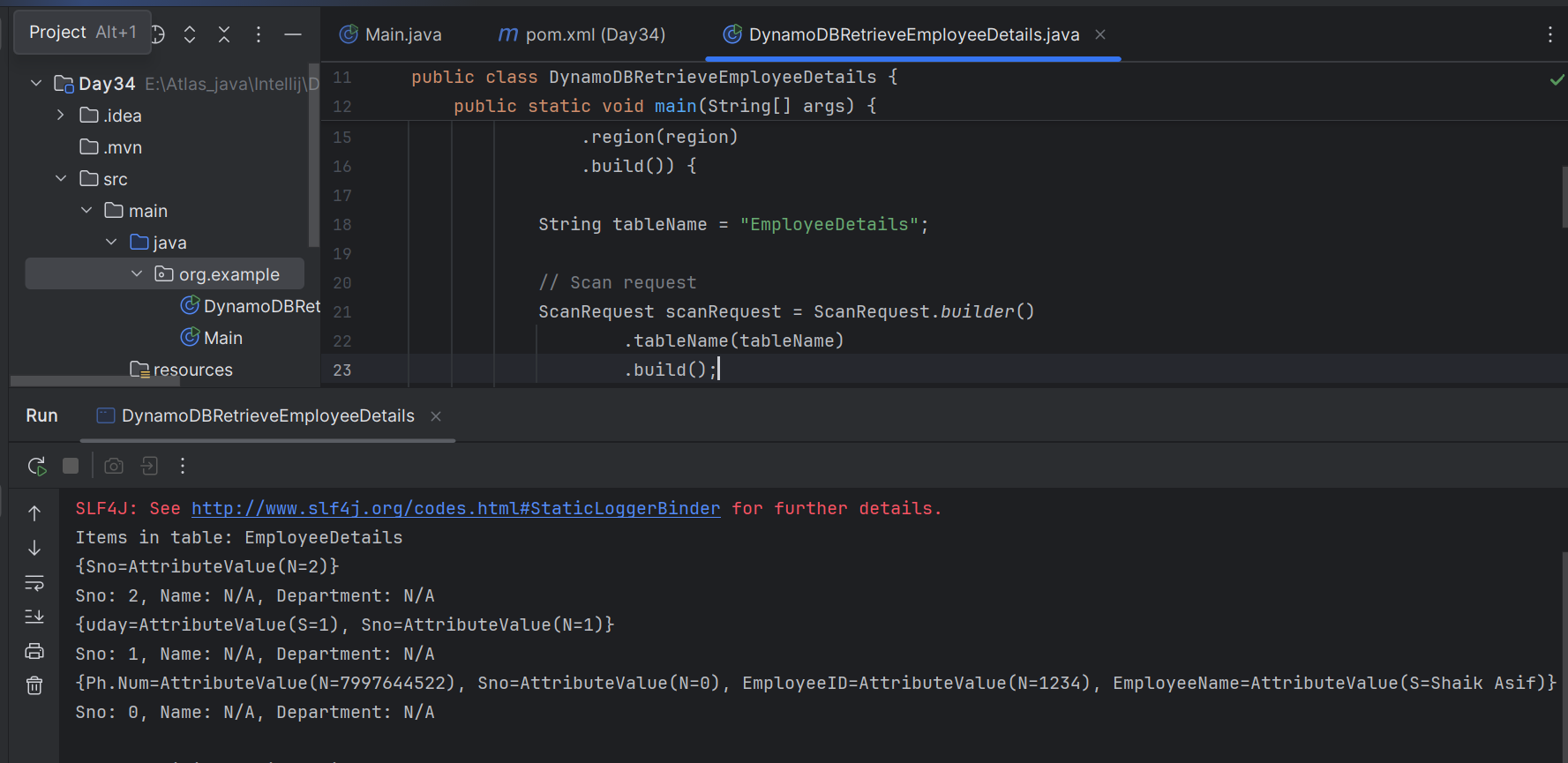
Login ID: iamasif

Name: Shaik Asif

Task 01:

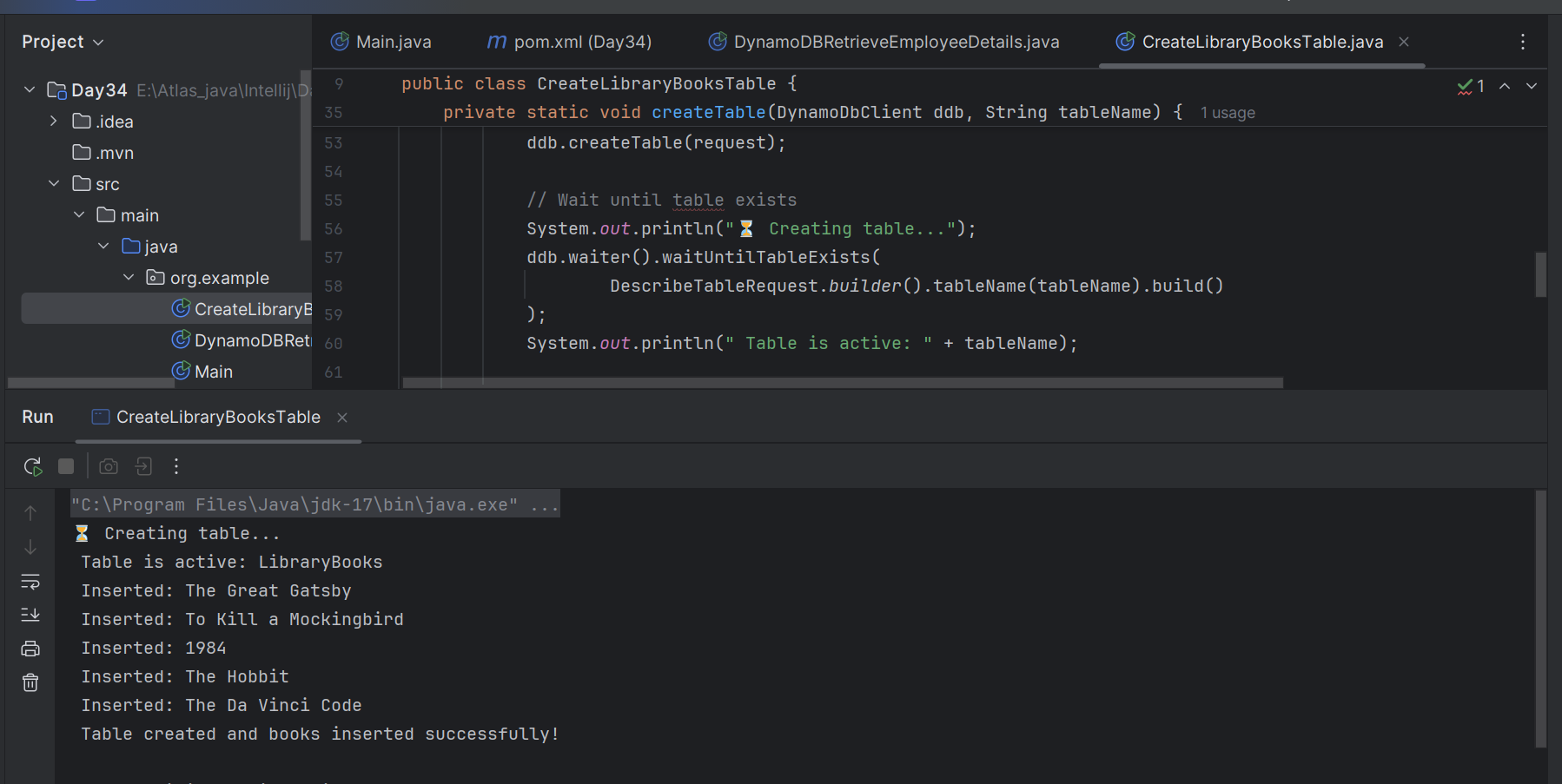
Create a java program to retrieve dynamodb ttable records and display..

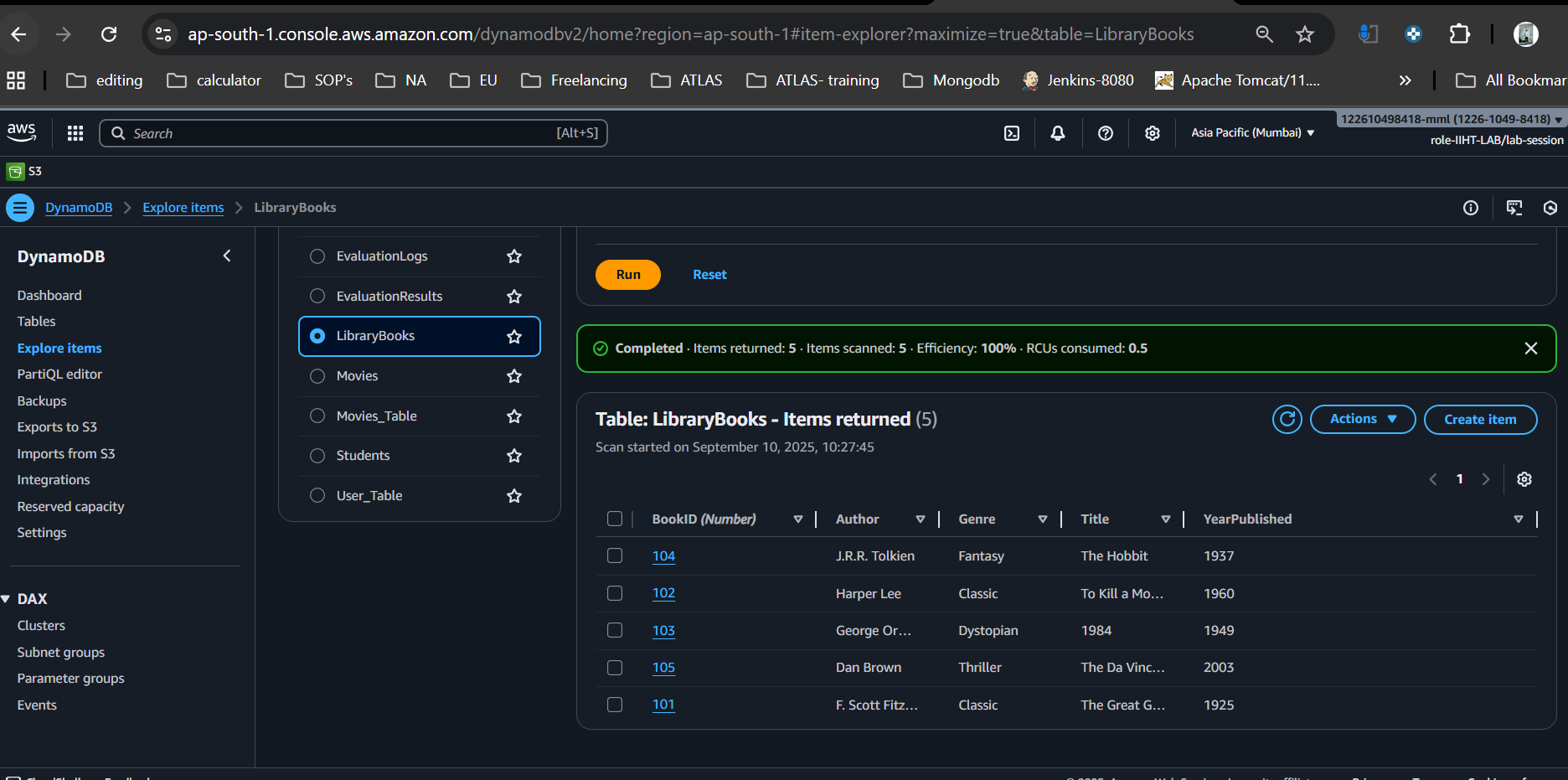
9.47 to 9.57



TASK 02:

Create a java program to create a table and insert 2 items .. the NOSQl workbench we should be able to see the table updated.





Task 03:

What key features of DynamoDB?

### **1. Fully Managed & Serverless**

* No need to manage servers, clusters, or scaling manually.
* AWS handles hardware provisioning, setup, configuration, replication, backups, and software patching.

### **2. High Performance at Scale**

* Supports **single-digit millisecond response times** consistently.
* Can scale to handle **millions of requests per second** automatically.

### **3. Flexible Data Model**

* **NoSQL database**: stores data as **key-value pairs** and **documents**.
* Tables consist of **items (rows)** and **attributes (columns)**.
* Schema-less: different rows in the same table can have different attributes.

### **4. Primary Key Options**

* Two types of primary keys:  
  1. **Partition Key only** (simple key)
  2. **Partition Key + Sort Key** (composite key for hierarchical/temporal data).

### **5. Secondary Indexes**

* **Global Secondary Index (GSI):** query data on any non-primary key.
* **Local Secondary Index (LSI):** query with an alternate sort key on the same partition key.

### **6. On-Demand or Provisioned Capacity**

* **On-Demand mode:** pay-per-request, auto-scales capacity.
* **Provisioned mode:** you define read/write capacity units (RCUs/WCUs) and use auto-scaling.

### **7. Global Tables (Multi-Region Replication)**

* Multi-active replication across AWS regions.
* Ensures **low-latency reads/writes globally** and high availability.

### **8. Durability & High Availability**

* Data is replicated **across 3 AZs (Availability Zones)** in a region.
* Provides **99.999% (five nines) availability** SLA.

### **9. Transactions**

* Supports **ACID transactions** (all-or-nothing operations) across multiple items and tables.

### **10. Event-Driven Integrations**

* Streams integration with **AWS Lambda** → automatically trigger functions on data changes.
* Useful for **real-time analytics, triggers, caching, notifications, etc.**

### **11. Security**

* Integrated with **AWS IAM** for access control.
* Supports **encryption at rest** (AWS KMS) and **in transit** (TLS).

### **12. Backup & Restore**

* On-demand backups and **point-in-time recovery (PITR)** for the last 35 days.

### **13. Cost Optimization**

* Pay only for the storage and requests you use.
* Options: **On-Demand** or **Provisioned Capacity** (with auto-scaling).

# **Summary**

Amazon DynamoDB is:

* **Serverless**
* **Highly available**
* **Low latency**
* **Scalable**
* **Flexible schema**
* **Integrated with AWS ecosystem**

Task 04:

What are the advantages and disadvantages of DynamoDB?

| **Advantages** | **Disadvantages** |
| --- | --- |
| **Fully Managed (Serverless)** – No servers, patching, or replication to manage | **Complex Pricing** – Costs can grow quickly with high RCU/WCU usage |
| **Highly Scalable** – Handles millions of requests per second | **Limited Query Flexibility** – Only efficient for key lookups; requires GSIs/LSIs for other queries |
| **Low Latency** – Single-digit millisecond performance | **No Joins or Aggregations** – Cannot perform SQL-like joins or GROUP BY; must use external tools |
| **Flexible Schema** – Schema-less, supports key-value & documents | **Item Size Limit** – Max item size is 400 KB |
| **Global Tables** – Multi-region replication with low-latency access | **Eventual Consistency in Multi-Region** – Strong consistency only within a single region |
| **Durability & Availability** – Replicates across 3 AZs, 99.999% SLA | **Schema Design Complexity** – Poor design may cause hot partitions |
| **Integrated with AWS** – Works with Lambda, API Gateway, S3, etc. | **Vendor Lock-In** – Proprietary AWS service; migration is hard |
| **Security** – IAM fine-grained control, encryption at rest & transit | **Learning Curve** – Requires deep understanding of keys, partitions, indexes |
| **ACID Transactions** – Multi-item, multi-table transactions supported | – |
| **Backup & PITR** – On-demand backups and point-in-time recovery (35 days) | – |

Task5:

Where do we use DynamoDB (use cases of DynamoDB)

# **Use Cases of DynamoDB**

### **1. E-Commerce Applications**

* Store product catalogs, inventory, pricing, and customer orders.
* Example: Flipkart, Amazon use DynamoDB for **shopping cart data**.
* Handles **millions of users** with **low latency**.

**2. Gaming**

* Store player profiles, leaderboards, in-game progress, and session data.
* Supports **real-time updates** and **high request rates**.
* Example: Mobile game backends where thousands of players update scores simultaneously.

### **3. IoT (Internet of Things)**

* Store sensor data, device logs, and IoT event streams.
* Handles **large volumes of time-series data**.
* Often paired with **AWS IoT Core + DynamoDB + Lambda**.

**4. Social Media / Messaging**

* Store user profiles, friend connections, messages, likes, comments.
* Sub-millisecond response → good for **real-time apps**.
* Example: Chat apps, news feeds, follower systems.

### **5. Financial Services**

* Store transactions, account balances, fraud detection logs.
* Supports **ACID transactions** for reliability.
* Example: Banking apps, payment processing systems.

### **6. Serverless Applications**

* DynamoDB pairs well with **AWS Lambda** and **API Gateway**.
* Used for building **scalable serverless microservices**.
* Example: Order tracking, ticket booking, event-driven apps.

### **7. Content Management & Personalization**

* Store articles, blog posts, media metadata, and personalized recommendations.
* Handles **structured + semi-structured data**.
* Example: News portals, video streaming (Netflix uses it for metadata).

### **8. Session Management**

* Store user session state in web/mobile apps.
* Fast read/write makes it ideal for **login tokens, shopping carts, preferences**.

### **9. Real-Time Analytics**

* Store logs, metrics, clickstreams, and process them via **Kinesis + DynamoDB Streams + Lambda**.
* Enables dashboards and monitoring in **real-time**.

### **10. Healthcare & Life Sciences**

* Store patient data, medical IoT records, appointment scheduling.
* HIPAA-eligible with **encryption & IAM security**.

# **Summary of Use Cases (Quick List)**

* **E-commerce** → product catalogs, shopping carts
* **Gaming** → player data, leaderboards
* **IoT** → sensor data, logs
* **Social Media** → feeds, likes, chats
* **Finance** → transactions, balances
* **Serverless apps** → with Lambda + API Gateway
* **Content management** → blogs, video metadata
* **Session management** → authentication, carts
* **Analytics** → logs, clickstreams
* **Healthcare** → patient & medical records

Task 06:

What is DynamoDBMapper?

DynamoDBMapper is a high-level, object persistence API provided by the AWS SDK for Java.

It lets you map Java classes (POJOs) to DynamoDB tables, so you can work with DynamoDB using Java objects instead of raw requests.

## **Key Features of DynamoDBMapper**

1. Annotation-based Mapping  
   1. Use annotations like @DynamoDBTable, @DynamoDBHashKey, and @DynamoDBAttribute to map Java classes to DynamoDB tables.
2. CRUD Operations  
   1. Provides easy methods:  
      * save() → Insert/Update an item
      * load() → Retrieve an item by key
      * delete() → Delete an item
      * scan() / query() → Retrieve multiple items
3. Automatic Conversion  
   1. Converts between Java types and DynamoDB types (String, Number, Boolean, etc.).
4. Reduces Boilerplate  
   1. You don’t need to manually build PutItemRequest, GetItemRequest, etc.

## **Advantages of DynamoDBMapper**

* Simple & clean API (no manual request building).
* Reduces code → easier to maintain.
* Maps directly between Java objects and DynamoDB tables.

## **When to Use DynamoDBMapper**

* When working with DynamoDB in Java applications.
* For projects that need object-oriented abstraction instead of raw requests.
* Useful in enterprise apps, microservices, and serverless backends.

DynamoDBMapper = Java ORM for DynamoDB → lets you work with DynamoDB tables as if they were normal Java objects.

**Task 07:**

What are DynamoDb Projections?

In DynamoDB, a projection determines which attributes from the base table are copied into a secondary index.

When you create a Global Secondary Index (GSI) or a Local Secondary Index (LSI), you must choose what attributes should be available in that index.  
 This selection is called a Projection.

## **Why Do We Need Projections?**

* Secondary indexes are stored separately from the base table.
* By default, not all attributes from the main table are copied (to save storage & cost).
* You decide which attributes you want to project into the index, so queries on the index can retrieve them directly.

**Types of Projections in DynamoDB**

| **Projection Type** | **What it Includes** | **When to Use** |
| --- | --- | --- |
| **KEYS\_ONLY** | Only the **index keys** + the **primary key of the base table** | Small storage, fast lookup. Use when you only need keys for joins/queries. |
| **INCLUDE** | Index keys + base table primary key + **specific non-key attributes** you choose | Optimized for queries that need a few extra attributes. |
| **ALL** | All attributes from the base table are copied into the index | Simplest but costly (more storage + write overhead). Useful when you query many attributes. |

## **Advantages of Projections**

* Reduce storage cost (by projecting fewer attributes).
* Optimize query performance (queries can be served directly from the index).
* Flexibility: choose only what’s needed.

**Projection = attributes copied from base table → index**.

Types:

* KEYS\_ONLY → smallest, fastest.
* INCLUDE → add selected attributes.
* ALL → full copy of table attributes.

**Task 08:**

How can you say that DYnamoDB prevents data loss?

### **1. Multi-AZ Replication (High Availability & Durability)**

* Every item written to a DynamoDB table is synchronously replicated across 3 Availability Zones (AZs) in an AWS region.
* If one AZ goes down, your data is still safe and available in the other 2 AZs.
* This ensures 99.999% availability and prevents single-point-of-failure data loss.

### **2. Strong Consistency Option**

* By default, DynamoDB offers eventual consistency (faster but might show slightly stale data).
* If you want absolute data safety, you can request strongly consistent reads, ensuring you always get the latest data written to all replicas.

### **3. Durable Write Acknowledgments**

* A write operation is only acknowledged after being stored in multiple AZs.
* This prevents data loss in case of immediate hardware or network failures.

### **4. Backups & Point-in-Time Recovery (PITR)**

* DynamoDB provides on-demand backups (full snapshots of your tables).
* PITR (Point-in-Time Recovery) lets you restore your table to any second within the last 35 days.
* This protects against accidental deletes, overwrites, or corruption.

### **5. DynamoDB Streams (Change Data Capture)**

* Streams record every data modification (insert, update, delete).
* You can use this to recover lost data or replicate data to another system in near real-time.

### **6. ACID Transactions**

* DynamoDB supports ACID transactions across multiple items and tables.
* Ensures all-or-nothing execution (no partial writes), preventing logical data corruption.

### 7. Global Tables (Cross-Region Replication)

* With Global Tables, DynamoDB automatically replicates data across multiple AWS regions.
* Even if an entire region fails, your data is available in another region.

# Summary

DynamoDB prevents data loss through:

* Multi-AZ replication (3 copies per region)
* Strongly consistent reads
* Durable write acknowledgments
* Backups & Point-in-Time Recovery (PITR)
* Streams for data replay/recovery
* ACID transactions
* Global tables for cross-region durability

**Task 09:**

What is in-place atomic updates means ? does dynamoaDB supports?

In-place atomic update means updating a part of an item directly in the database without rewriting the entire item, and the update happens atomically (all-or-nothing).

* In-place → Only the specific attribute is modified, not the whole row.
* Atomic → The operation is safe from race conditions (if multiple users update the same item at the same time, DynamoDB ensures correctness).

In-place atomic update = update specific attributes safely without rewriting the whole item.

Yes, DynamoDB supports it via:

* UpdateItem API with update expressions
* Atomic counters (ADD operation)
* Conditional writes

12.22 to 12.28

Task 10:

What are Streams in DynamoDB?

**DynamoDB Streams** is a feature that **captures a time-ordered sequence of item-level changes** (insert, update, delete) in a DynamoDB table.

* Think of it as a **transaction log** for your DynamoDB table.
* Each change in the table is recorded in the stream for up to **24 hours**.
* Other AWS services (like **Lambda**) can consume the stream to take automatic actions.

# **What Kind of Changes are Captured?**

For each item change, a stream record is generated that can include:

* INSERT → New item added
* MODIFY → Existing item updated
* REMOVE → Item deleted

# **Stream Views (Types of Data You Can Capture)**

When you enable a stream, you choose the view type:

| Stream View Type | What It Contains |
| --- | --- |
| KEYS\_ONLY | Only the primary key attributes of changed items |
| NEW\_IMAGE | The entire item after the change |
| OLD\_IMAGE | The entire item before the change |
| NEW\_AND\_OLD\_IMAGES | Both the item before and after the change |

# **How Streams Work**

1. You enable a stream on a table.
2. Every insert, update, or delete generates a stream record.
3. The stream record is available for 24 hours.
4. Applications (like AWS Lambda, Kinesis, EC2) can process these records.

# **Example Use Cases of DynamoDB Streams**

1. Triggering Lambda Functions  
   * Send a notification/email when a new order is placed.
   * Update a cache when table data changes.
2. Audit Logging  
   * Maintain a history of changes (before & after values).
3. Replication  
   * Keep two DynamoDB tables (even across regions) in sync.
4. Analytics  
   * Push change events into Kinesis / Redshift for real-time analytics.

DynamoDB Streams = ordered log of item-level changes in a table.

Retains changes for 24 hours.

Supports 4 stream views (keys only, new, old, both).

Commonly used for triggers, replication, auditing, real-time analytics.

Task 11:

What are DynamoDb Pricing Tiers?

Amazon DynamoDB pricing is mainly based on capacity mode, storage, and optional features.

1. Capacity Modes

| **Mode** | **How It Works** | **When to Use** |
| --- | --- | --- |
| On-Demand | Pay per request. You don’t need to specify capacity units. DynamoDB automatically scales to your workload. | Unpredictable traffic, spiky workloads, dev/test environments. |
| Provisioned | You specify Read Capacity Units (RCUs) and Write Capacity Units (WCUs) in advance. Can enable Auto Scaling. | Predictable traffic, steady workloads, cost-sensitive apps. |

## 2. **Free Tier**

* First 25 GB of storage per month is free.
* First 25 WCUs and 25 RCUs are free for 12 months.

## 3. **Storage Pricing**

* You pay per GB stored per month.
* As of 2025: $0.25 per GB-month (in most regions).

4. **Optional Features Pricing**

* Streams → Charged per read request unit.
* Global Tables → Charged for replicated writes in each region.
* DAX (DynamoDB Accelerator) → Charged per node-hour.
* Backup & Restore → Charged per GB of backup.
* Point-in-Time Recovery (PITR) → Charged per GB of storage per month.

## 5. **Data Transfer**

* Within same region → Free.
* Cross-region replication (Global Tables) → Charged per GB transferred.

# Example Costing (Simplified)

Let’s say:

* Table has 50 GB of data
* Uses on-demand mode
* Handles 1 million reads + 1 million writes per month

Rough cost would include:

* Storage = 50 GB × $0.25 = $12.50
* Reads = ~ $1.25
* Writes = ~ $1.25
* Total ≈ $15 per month (excluding extras like Streams/PITR).

# **Summary**

DynamoDB Pricing Tiers are based on:

1. Capacity Modes → On-Demand (pay per request) OR Provisioned (RCU/WCU).
2. Storage → Per GB per month.
3. Optional Features → Streams, Global Tables, DAX, PITR, Backups.
4. Data Transfer → Cross-region costs extra.
5. Free Tier → 25 GB + 25 WCUs/RCUs free for 12 months.

12.09 to 12.33

Task 12:

Do you have any max limit for Item Size in DynamoDb? If so what is it?

# **Maximum Item Size in DynamoDB**

* The maximum size of a single item in DynamoDB is 400 KB (kilobytes).
* This includes:  
  + Attribute names
  + Attribute values
  + Binary data
  + Overhead for JSON formatting

In other words, the sum of all attributes in an item cannot exceed 400 KB.

# **Implications**

* You cannot store very large objects (e.g., big images, PDFs, videos) directly in DynamoDB.
* Best practice: store large objects in Amazon S3, and keep only a reference (URL/ID) in DynamoDB.

# **Example**

{

"UserID": 123,

"Name": "Alice",

"Bio": "Hello World...",

"ProfilePic": "https://s3.amazonaws.com/bucket/user123.jpg"

}

**Not Allowed:**

Storing the entire image binary in DynamoDB if it exceeds 400 KB.

Maximum item size in DynamoDB = 400 KB.

Includes all attributes (keys + values).

For large data → use S3 for storage + DynamoDB for metadata/indexing.

**Task 13:**

At a max of how many GSI’s Global Secondary Indexes can you create in a table?

# **Maximum Number of GSIs in DynamoDB**

* You can create up to 20 Global Secondary Indexes (GSIs) per table.

# **Other Index Limits (for context)**

* Local Secondary Indexes (LSIs): Maximum 5 per table.
* Global Secondary Indexes (GSIs): Maximum 20 per table.
* Both GSIs and LSIs can be created at table creation (LSIs only) or later (GSIs can be added anytime).

**Summary**

* Maximum GSIs per table = 20
* Maximum LSIs per table = 5

**Task 14:**

What is DynamoDb Accelarator?

DynamoDB Accelerator (DAX) is a fully managed, in-memory cache for DynamoDB that delivers microsecond response times for read-heavy workloads.

Think of it as Redis/Memcached built into DynamoDB, managed by AWS.

# **Key Features of DAX**

1. In-Memory Caching  
   * Caches DynamoDB read results in memory.
   * Reduces response times from milliseconds → microseconds.
2. Managed Service  
   * AWS handles cluster setup, replication, patching, and scaling.
3. Compatible API  
   * Uses the same DynamoDB SDK API calls (GetItem, Query, Scan).
   * Just configure the SDK to use a DAX client → no major code changes.
4. Clustered & Highly Available  
   * Runs as a cluster (1–10 nodes).
   * Supports multi-AZ replication for fault tolerance.
5. Security  
   * Supports VPC integration, encryption at rest, and IAM policies.

# **How DAX Works**

* When your app makes a GetItem or Query request:  
  1. The request first checks the DAX cache.
  2. If data is cached (cache hit), it returns immediately (microseconds).
  3. If not (cache miss), it fetches from DynamoDB, stores in cache, and then returns.

# **When to Use DAX**

Good for:

* Read-heavy workloads (e.g., recommendation engines, leaderboards, session stores).
* Applications needing sub-millisecond latency.
* When you want a managed cache without deploying Redis/Memcached yourself.

**Not ideal for:**

* Write-heavy workloads (since writes still go directly to DynamoDB).
* Apps needing very complex queries (DAX only accelerates existing DynamoDB API calls).

# **Example Use Case**

* An e-commerce app where product catalog data is read thousands of times per second but updated infrequently.
* With DAX, customers see product details instantly without hitting DynamoDB every time.

# 

# Summary

* DynamoDB Accelerator (DAX) = AWS-managed in-memory cache for DynamoDB.
* Reduces latency to microseconds for reads.
* Best for read-heavy, high-traffic workloads.
* Fully managed, highly available, and API-compatible.

**Tsk 15:**

What are DynamoDB Global tables?

DynamoDB Global Tables provide multi-region, multi-master replication for your DynamoDB tables.  
 This means:

* Your data is automatically replicated across multiple AWS regions.
* You can read and write to the table from any region.
* DynamoDB handles synchronous replication behind the scenes.

# **Key Features**

1. Multi-Region Replication  
   * Data is automatically copied across the regions you choose.
2. Multi-Master Writes  
   * You can write to the table in any region, not just one "primary" region.
3. Low-Latency Access  
   * Users around the world can read/write to the nearest AWS region with low latency.
4. Automatic Conflict Resolution  
   * If the same item is updated in two regions at the same time, DynamoDB uses last-writer-wins (based on timestamps).
5. Fully Managed  
   * No custom replication setup — AWS manages it for you.

# **Use Cases of Global Tables**

* Disaster Recovery → If one region goes down, your app can switch to another region without downtime.
* Global Applications → Social media, gaming, or e-commerce platforms serving users worldwide.
* Low-Latency Multi-Region Access → Users in India, US, and Europe can access the same app with minimal delay.
* Compliance → Keep data in multiple regions for regulations.

**Example**

Suppose you create a Global Table called Orders in ap-south-1 (Mumbai) and us-east-1 (Virginia):

* A user in India places an order → written to Mumbai.
* DynamoDB automatically replicates that order to Virginia.
* A US-based support agent can instantly see the order in Virginia region.

# **Pricing Notes**

* You pay for:  
  + Writes in each region
  + Data replication traffic between regions
  + Normal DynamoDB table costs (storage, reads/writes)

# **Summary**

* Global Tables = Multi-Region, Multi-Master DynamoDB tables.
* Automatic replication across regions.
* Provides high availability, disaster recovery, low-latency global access.
* Conflict resolution = last writer wins.

12.49 to 12.54

**Task 16:**

What are indexes and Secondary indexes in DynamoDB?

In DynamoDB, an index is a data structure that lets you query data efficiently on attributes other than the primary key.

* Every table must have a primary index (the primary key).
* You can also define secondary indexes for alternative query patterns.

## **Primary Index**

This is defined when you create a table. Two types:

1. Partition Key only (Simple Primary Key)  
   * Example: UserID
   * Each item is uniquely identified by this key.
2. Partition Key + Sort Key (Composite Primary Key)  
   * Example: UserID + OrderDate
   * Allows multiple items with the same partition key, sorted by sort key.

# **Secondary Indexes in DynamoDB**

A secondary index allows you to query data using attributes other than the table’s primary key.  
 There are two types:

### **1️. Local Secondary Index (LSI)**

* Shares the same partition key as the base table.
* Can use a different sort key.
* Must be defined at table creation time (cannot add later).
* Maximum 5 LSIs per table.

**Example:**  
 Base table → Partition key: UserID, Sort key: OrderDate  
 LSI → Partition key: UserID, Sort key: OrderAmount  
 Lets you query a user’s orders sorted by amount instead of date.

### **2️. Global Secondary Index (GSI)**

* Can have a different partition key and sort key from the base table.
* Can be created anytime (not just at table creation).
* Maximum 20 GSIs per table.

**Example:**  
 Base table → Partition key: UserID  
 GSI → Partition key: Email  
 Lets you query users directly by email.

# **Projections in Indexes**

When creating a secondary index, you decide what attributes are projected (copied from the main table to the index):

* KEYS\_ONLY → Only index + primary key attributes.
* INCLUDE → Keys + specific extra attributes.
* ALL → All attributes.

# **Comparison: LSI vs GSI**

| **Feature** | **Local Secondary Index (LSI)** | **Global Secondary Index (GSI)** |
| --- | --- | --- |
| Partition Key | Same as base table | Can be different |
| Sort Key | Different from base table | Optional, can be different |
| Create Time | Must be created with table | Can be created anytime |
| Limit | Max 5 per table | Max 20 per table |
| Use Case | Query within the same partition | Query across all partitions |

**Task 17:**

What are Hot Keys and Hot Partitions?

## What is a Hot Partition?

A Hot Partition is when an entire partition (the storage node for multiple keys) gets more traffic than others.

* DynamoDB partitions data across multiple physical storage nodes.
* If one partition is accessed too heavily compared to others, it’s a hot partition.

Problem: Throughput for that partition maxes out, even though other partitions are underutilized.

## **Example**

### **Hot Key Example:**

Table: Orders  
 Partition Key = CustomerID

* If one customer (CustomerID = 12345) generates 90% of traffic, then that single key is "hot".
* This causes throttling even though table capacity is high.

### **Hot Partition Example:**

* Suppose 100 customers are spread across 10 partitions.
* If 50 customers (and their traffic) are mapped to the same partition by hash function → that partition becomes "hot".

**Task 18:**

What are Table level operations and Item level operations in DynamoDB?

## **1. Table-Level Operations**

These are operations that act on the entire table or its structure, not just a single item.

### **Examples:**

1. CreateTable → Create a new DynamoDB table with schema and capacity.
2. DeleteTable → Delete an entire table (and all its items).
3. UpdateTable → Modify table settings (capacity mode, throughput, GSIs/LSIs).
4. DescribeTable → Get metadata (status, keys, indexes, throughput).
5. ListTables → List all tables in the account/region.

**Use Case:**

* When you’re setting up or managing DynamoDB infrastructure.

## **2. Item-Level Operations**

These are operations that act on individual items (rows) within a table.

### **Examples:**

1. PutItem → Insert a new item (or replace if key already exists).
2. GetItem → Retrieve a single item by primary key.
3. UpdateItem → Update attributes of an item (supports in-place atomic updates).
4. DeleteItem → Remove a specific item.
5. Query → Retrieve multiple items using partition key (and optional sort key).
6. Scan → Retrieve all items in a table (slower, costly).
7. BatchWriteItem → Insert/delete multiple items at once.
8. BatchGetItem → Retrieve multiple items from one/many tables.

**Use Case:**

* When your application needs to read, write, update, or delete specific records.

# **Comparison Table**

| **Category** | **Operation Examples** | **Scope** |
| --- | --- | --- |
| Table-Level | CreateTable, DeleteTable, UpdateTable, DescribeTable, ListTables | Entire table (schema/structure/metadata) |
| Item-Level | PutItem, GetItem, UpdateItem, DeleteItem, Query, Scan, BatchWriteItem, BatchGetItem | Individual items (rows of data) |

**Creating a table using DyanmoDBMapper:**

**Course.java:  
  
Program:**

package org.example;

import com.amazonaws.services.dynamodbv2.datamodeling.\*;

@DynamoDBTable(tableName = "Courses")

public class Course {

private String courseId;

private String courseName;

private String instructor;

private int durationWeeks;

private String level;

@DynamoDBHashKey(attributeName = "CourseID")

public String getCourseId() { return courseId; }

public void setCourseId(String courseId) { this.courseId = courseId; }

@DynamoDBAttribute(attributeName = "CourseName")

public String getCourseName() { return courseName; }

public void setCourseName(String courseName) { this.courseName = courseName; }

@DynamoDBAttribute(attributeName = "Instructor")

public String getInstructor() { return instructor; }

public void setInstructor(String instructor) { this.instructor = instructor; }

@DynamoDBAttribute(attributeName = "DurationWeeks")

public int getDurationWeeks() { return durationWeeks; }

public void setDurationWeeks(int durationWeeks) { this.durationWeeks = durationWeeks; }

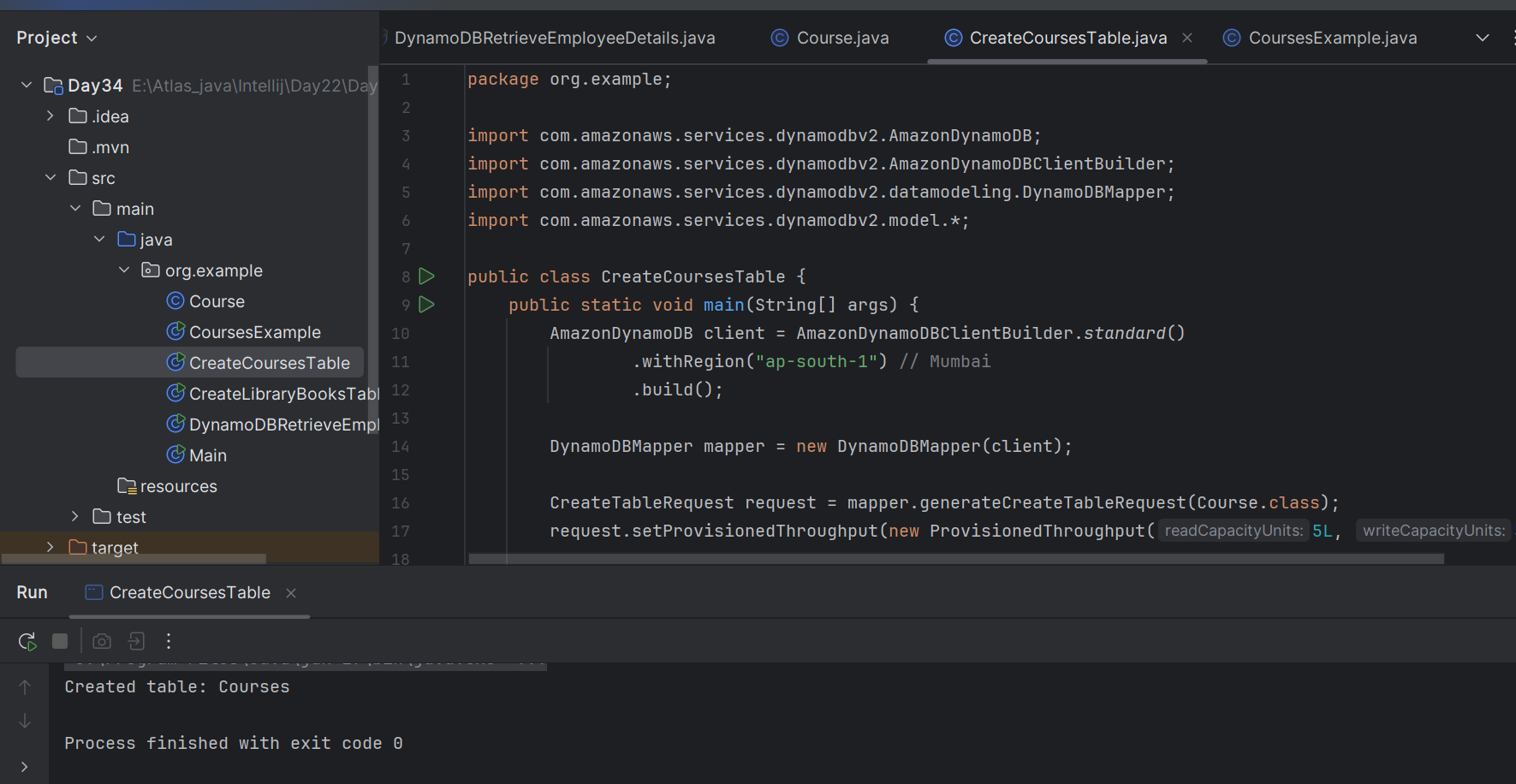
@DynamoDBAttribute(attributeName = "Level")

public String getLevel() { return level; }

public void setLevel(String level) { this.level = level; }

}

**Output:**



**Creating the Table with DynamoDBMapper**

**CreateCoursesTable.java**

**Program:**

package org.example;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;

import com.amazonaws.services.dynamodbv2.model.\*;

public class CreateCoursesTable {

public static void main(String[] args) {

AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()

.withRegion("ap-south-1") // Mumbai

.build();

DynamoDBMapper mapper = new DynamoDBMapper(client);

CreateTableRequest request = mapper.generateCreateTableRequest(Course.class);

request.setProvisionedThroughput(new ProvisionedThroughput(5L, 5L));

try {

client.createTable(request);

System.out.println(" Created table: " + request.getTableName());

} catch (ResourceInUseException e) {

System.out.println(" Table already exists: " + request.getTableName());

}

}

}

**Insert and Retrieve Data Using DynamoDBMapper**

**CreateCoursesTable.java**

package org.example;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;

import com.amazonaws.services.dynamodbv2.model.\*;

public class CreateCoursesTable {

public static void main(String[] args) {

AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()

.withRegion("ap-south-1") // Mumbai

.build();

DynamoDBMapper mapper = new DynamoDBMapper(client);

CreateTableRequest request = mapper.generateCreateTableRequest(Course.class);

request.setProvisionedThroughput(new ProvisionedThroughput(5L, 5L));

try {

client.createTable(request);

System.out.println(" Created table: " + request.getTableName());

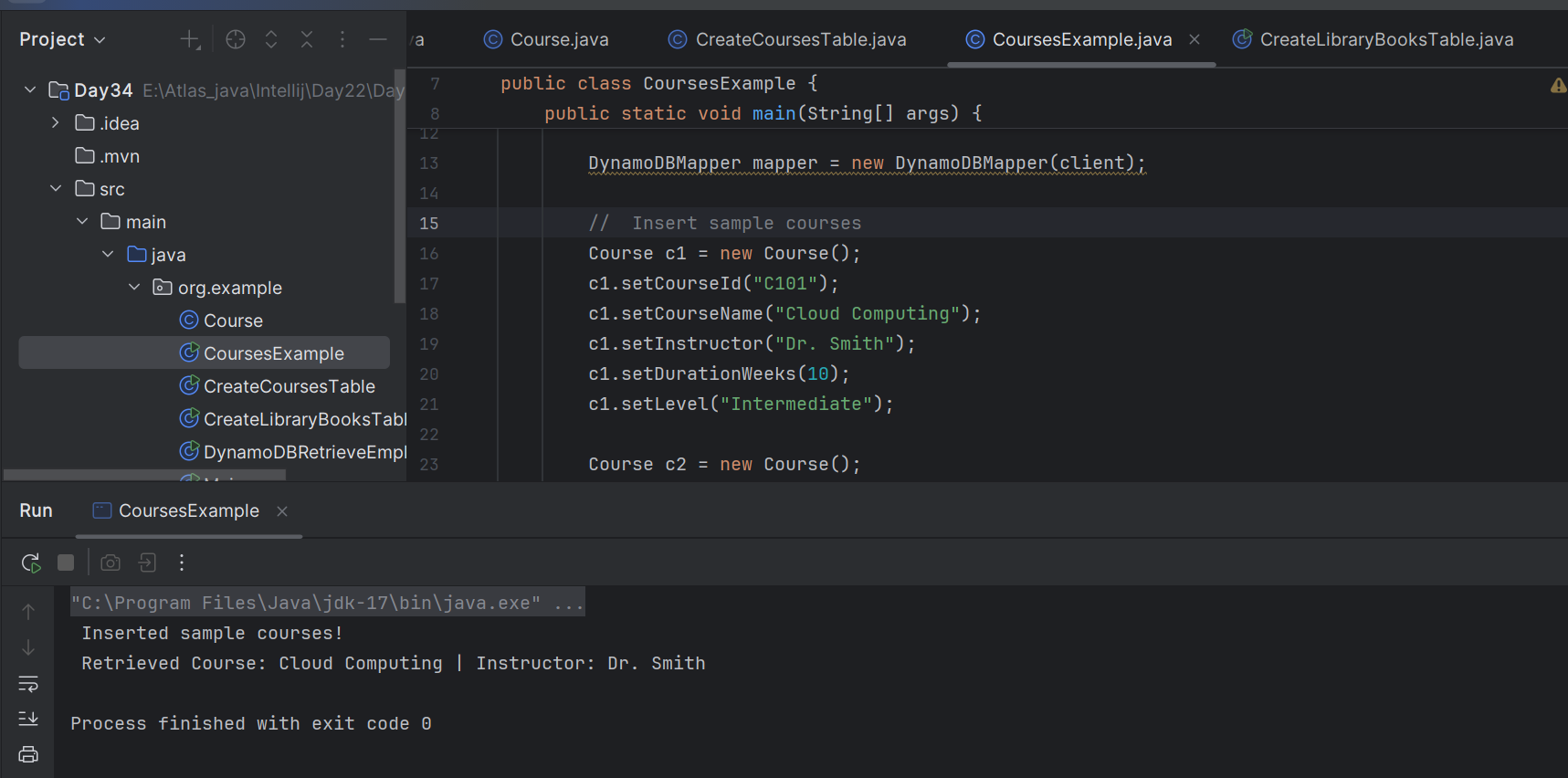
} catch (ResourceInUseException e) {

System.out.println(" Table already exists: " + request.getTableName());

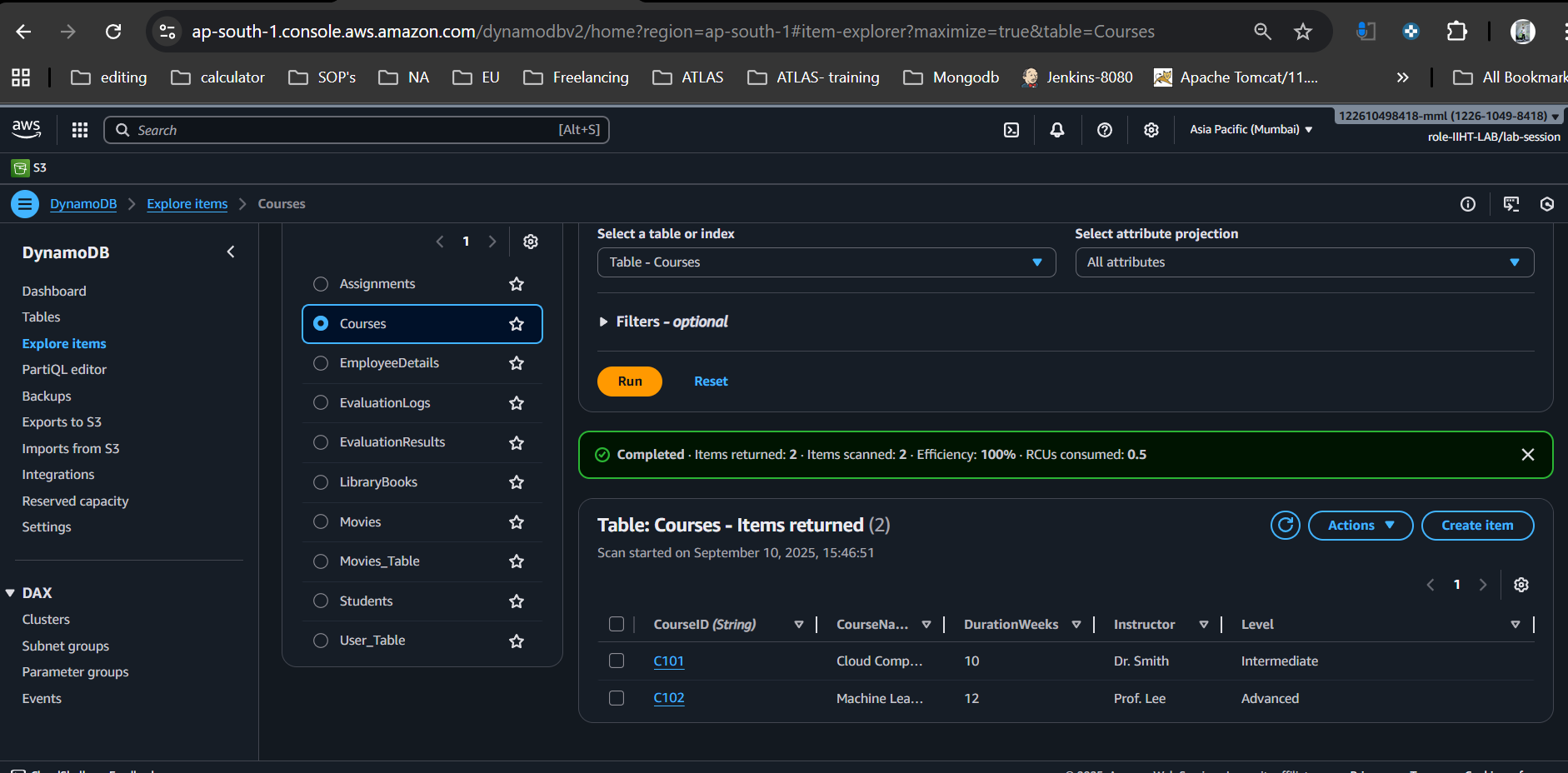
}

}

}



**AWS:**



INfo Box:

**<!-- SLF4J Simple Logger -->**

**<dependency>**

**<groupId>org.slf4j</groupId>**

**<artifactId>slf4j-simple</artifactId>**

**<version>2.0.16</version>**

**</dependency>**

By Sudipto to remove errors.. SLF4J logger error

DynamoDBMapper classes

