**day34\_107856406\_dsdipt\_sudipto\_10september2025**

**Employee Code:** 107856406

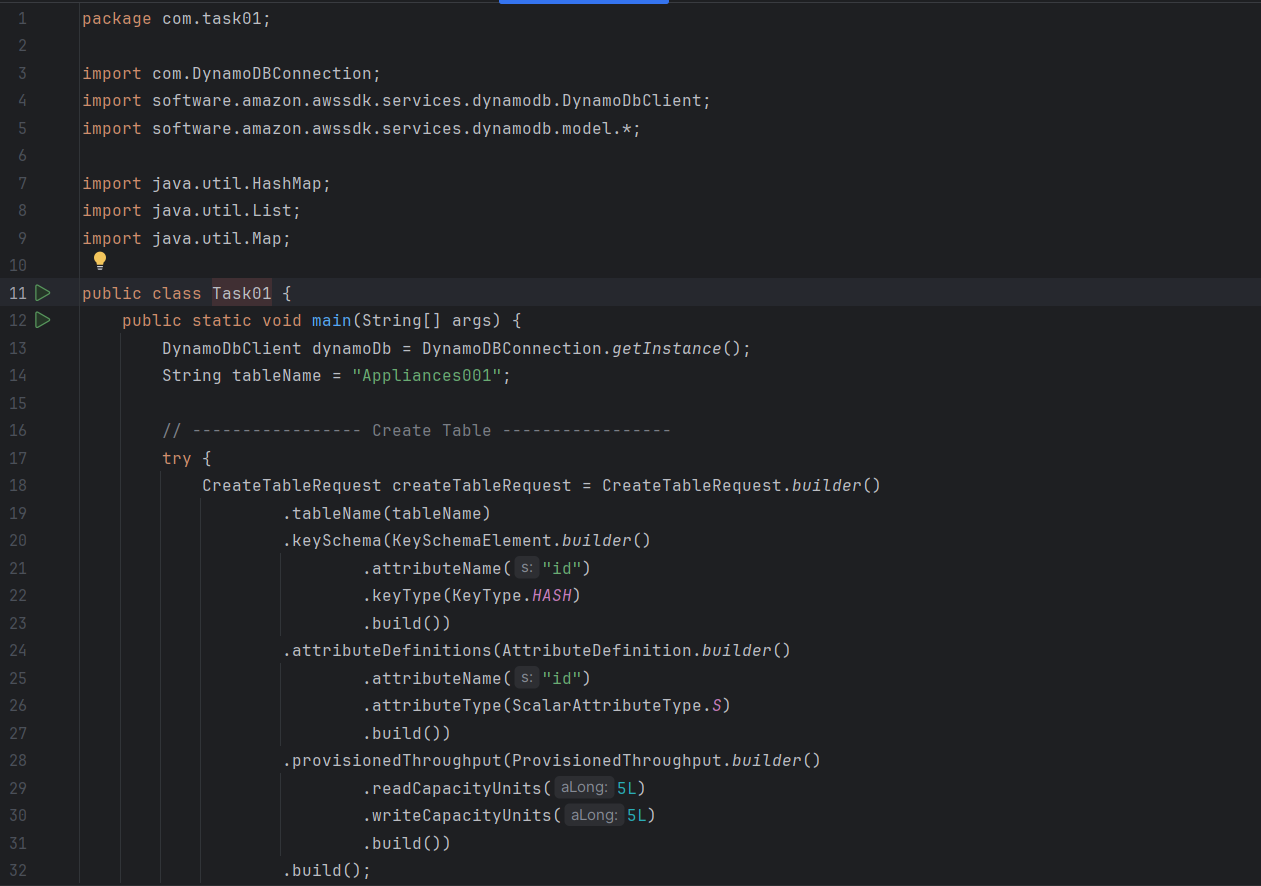
**Login ID:** dsdipt

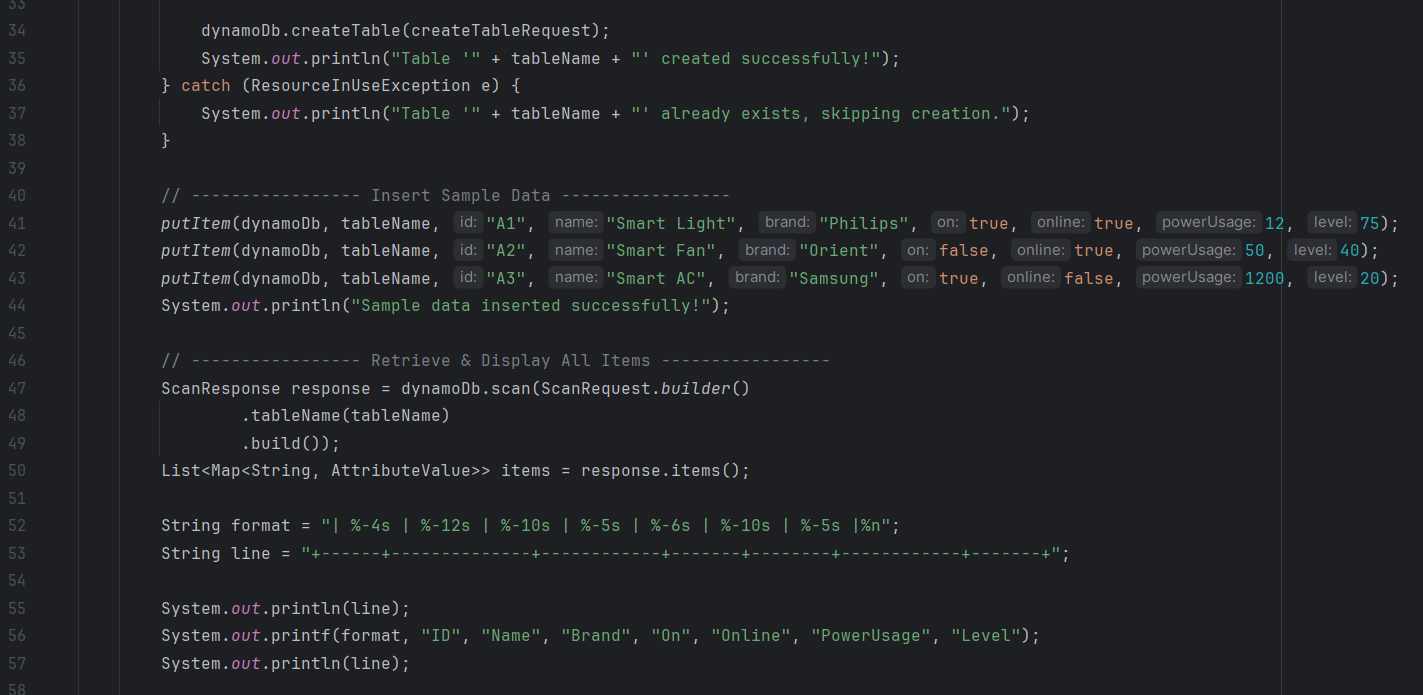
**Email:** dsdipt@amazon.com

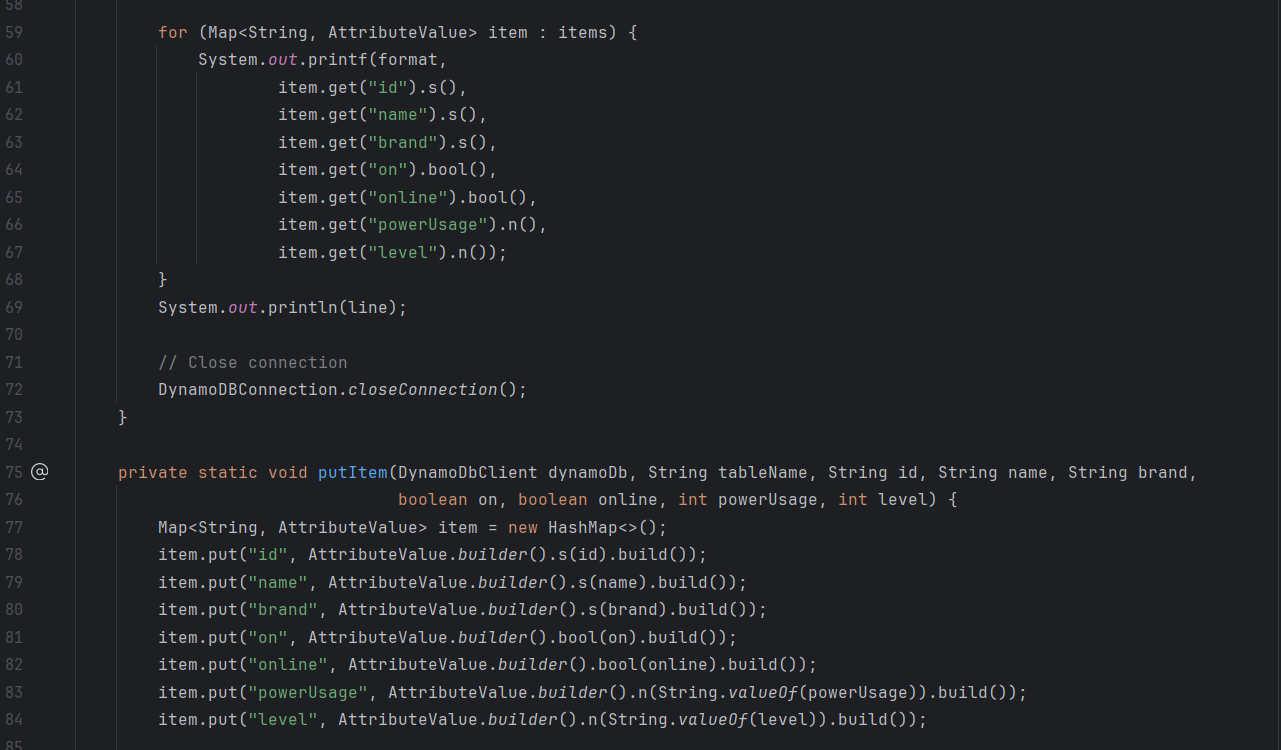
**Name:** Sudipto Das

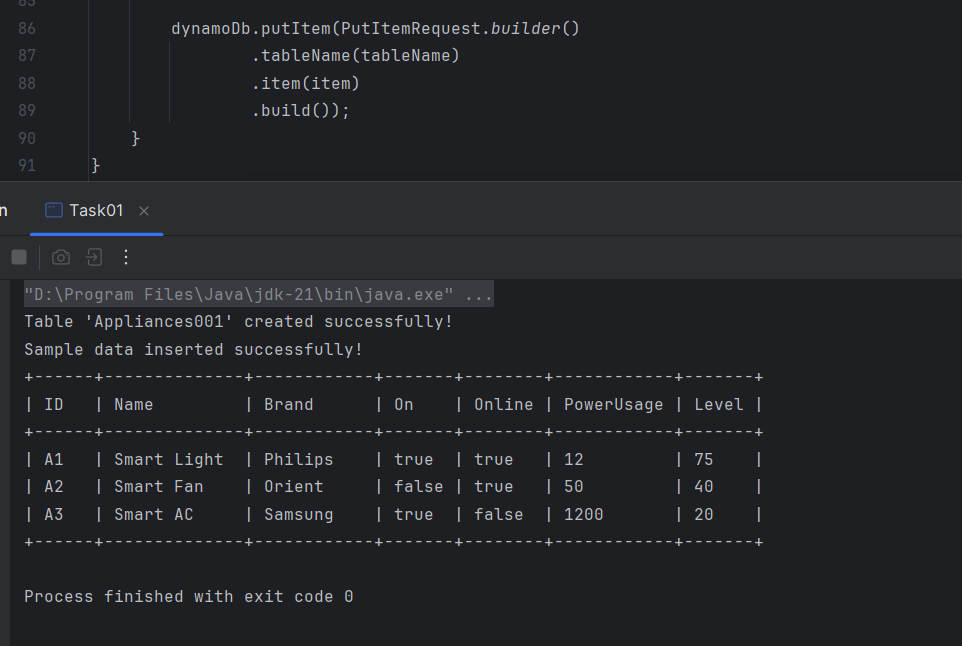
**Date:** 10 September 2025 (Day 34)

### ***Task 01: Create a java program to retrieve Dynamo DB table records and display.***

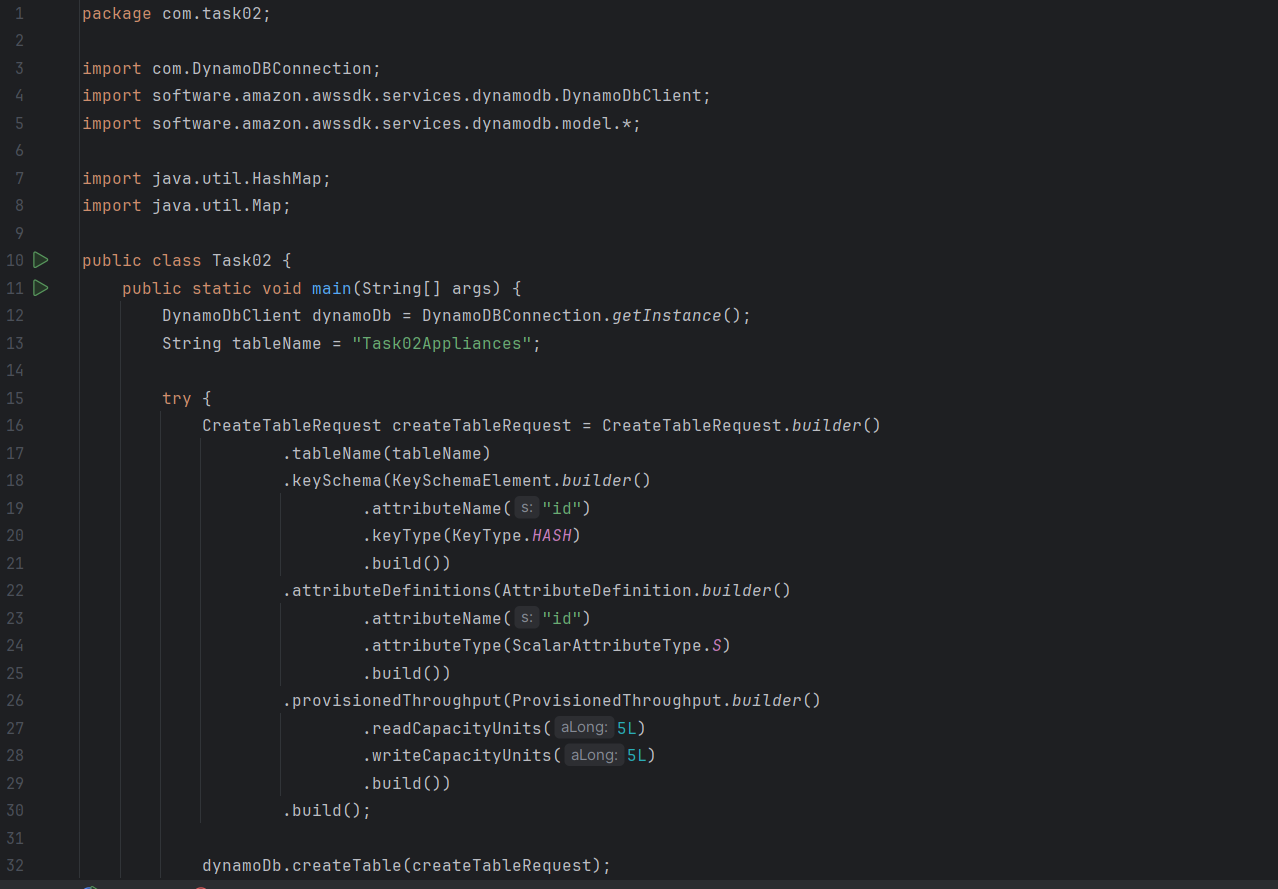


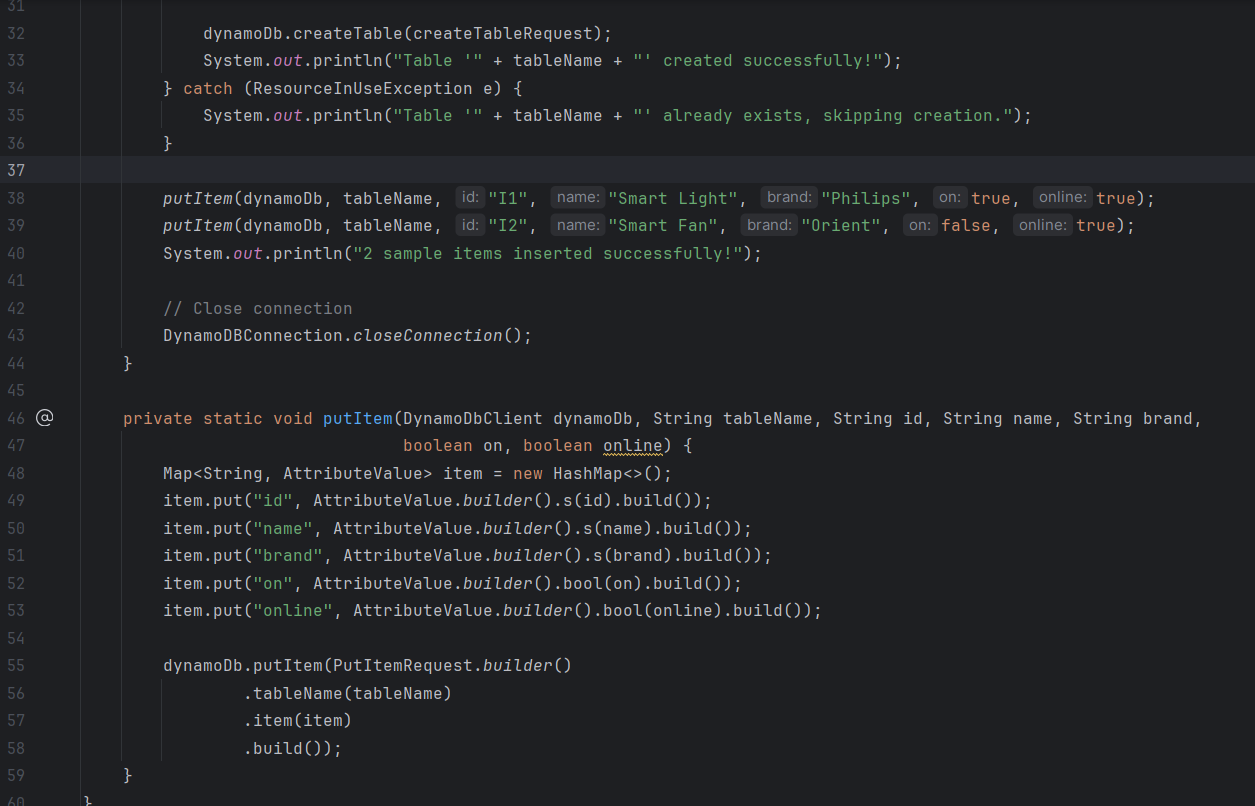
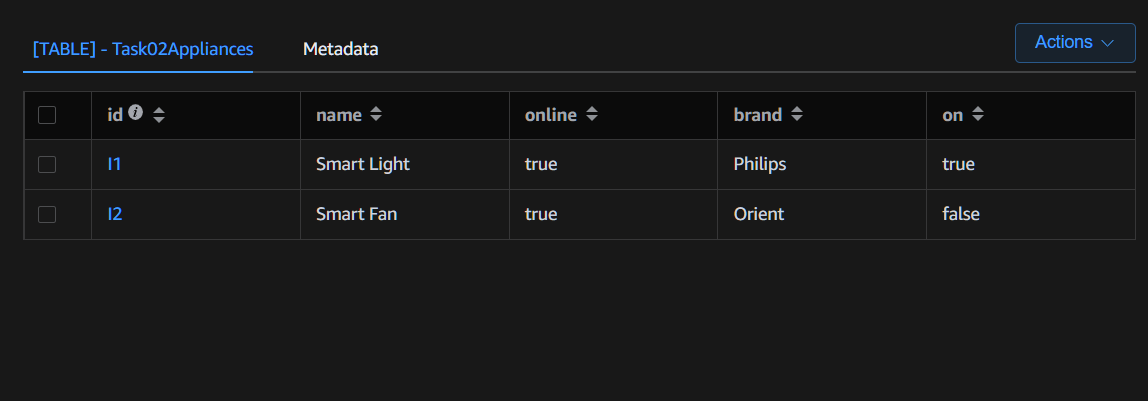






### ***Task 02: Create table & insert 2 items, should be visible in NoSQL Workbench.***



### ***Task 03: What are key features of DynamoDB?***

**Fully Managed**

No server setup, no scaling headaches. AWS handles the infrastructure. You just build apps

**High Performance at Scale**

Single-digit millisecond latency even at crazy scale 💨

**Flexible NoSQL Data Model**

Supports key-value and document data models → Super flexible schema design

**Auto Scaling**

DynamoDB auto adjusts capacity based on traffic patterns → Saves money + handles spikes

**Global Tables**

Multi-region replication for low latency global apps

**Built-in Security**

Fine-grained access control with IAM, encryption at rest & in transit

**Time-to-Live (TTL)**

Auto-delete expired items → Keeps your table clean & saves storage

**Streams & Triggers**

Change Data Capture → React to table changes in real-time with AWS Lambda

**On-Demand Mode**

Pay-per-request mode for unpredictable workloads → No need to provision capacity

### ***Task 04: What key features of DynamoDB?***

**Advantages of DynamoDB**

**Super Fast Performance**

* Single-digit millisecond latency, even with massive data
* Handles millions of requests per second

**Fully Managed Service**

* No need to worry about server setup, scaling, patching, or maintenance
* Auto scaling adjusts capacity based on usage

**Built-In Security**

* Fine-grained IAM policies
* Encryption at rest & in transit

**Flexible Data Model**

* Supports key-value and document storage
* No fixed schema → Great for evolving apps

**Time-to-Live (TTL)**

* Automatically deletes old data → Saves storage costs & keeps tables clean

**Streams & Triggers**

* Real-time data processing with DynamoDB Streams + AWS Lambda

**Global Tables**

* Multi-region replication → Ultra-low latency global apps

**Disadvantages of DynamoDB**

**Can Get Expensive**

* On-demand mode = easy but pricey for heavy workloads
* Provisioned capacity needs careful tuning or it wastes money

**Limited Query Flexibility**

* Queries only efficient on primary key or indexes
* No complex joins or aggregations like in relational DBs
* You often need to denormalize your data

**Size Limits**

* Max item size: 400 KB → You can’t shove huge objects in a single item
* Max 20 Global Secondary Indexes (GSIs) per table

Data Modeling Complexity

* Requires careful partition key + sort key design
* Bad partition keys → Hot partitions → Performance throttling

### ***Task 05: Where do we use DynamoDB?***

**Real-Time Web Apps**

Apps that need super fast, consistent reads and writes:  
 Chat apps (Slack-style), Social media feeds, Live notifications

**Global Applications**

Apps with users all over the world:  
 Gaming leaderboards, Global e-commerce catalogs (Amazon-level scale)  
 (use Global Tables for multi-region replication)

**IoT Applications**

Lots of devices sending small amounts of data constantly → Perfect for NoSQL and scalability  
Sensor data from smart homes, Wearable devices logging metrics

**Mobile Apps Backend**

Mobile apps that scale unpredictably → DynamoDB handles variable load perfectly  
 Todo apps, Location-based apps, User profiles

**Session Management**

Store ephemeral session data for apps in a super scalable way  
User login sessions, Shopping cart contents

**Analytics & Event Logging**

Store logs/events in a cheap and scalable way, often used together with DynamoDB Streams + Lambda for real-time processing  
 Clickstream tracking, App usage analytics

**E-Commerce Applications**

Store product catalogs, shopping carts, and order tracking  
Product metadata, order history, inventory tracking

**Microservices Data Store**

Lightweight microservices needing a fast key-value store → DynamoDB fits perfectly  
Config storage, Service discovery, API rate limiting

### ***Task 06: What is DynamoDB Mapper?***

**DynamoDBMapper** is a super convenient Java library that helps you map your Java objects (POJOs) to DynamoDB tables without writing a ton of manual code

* Automatically handles converting your Java objects into DynamoDB items and vice versa
* Saves you from writing low-level boilerplate for PutItem, GetItem, Query, Scan…
* Makes your code cleaner, easier to read, and faster to build

**How It Works**

**Annotate your class with @DynamoDBTable and mark keys with @DynamoDBHashKey, etc.**

@DynamoDbBean

public class Book {

private String id;

private String title;

private String author;

@DynamoDbPartitionKey

public String getId() { return id; }

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

public String getTitle() { return title; }

public void setTitle(String title) { this.title = title; }

public String getAuthor() { return author; }

public void setAuthor(String author) { this.author = author; }

}

**Use DynamoDBMapper to save or load objects**

DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()

.dynamoDbClient(dynamoDbClient)

.build();

DynamoDbTable<Book> bookTable = enhancedClient.table("Books", TableSchema.fromBean(Book.class));

// Save a book

Book book = new Book();

book.setId("1");

book.setTitle("1984");

book.setAuthor("George Orwell");

bookTable.putItem(book);

// Retrieve a book

Book savedBook = bookTable.getItem(r -> r.key(k -> k.partitionValue("1")));

### ***Task 07: What are Dynamo DB Projections?***

In DynamoDB, a **Projection** defines which attributes (columns) from a table are copied into a **Secondary Index** (like a Global Secondary Index, GSI).

**Projections help by:**

* Reducing data transfer
* Speeding up queries
* Lowering costs

**Types of Projections**

1. **ALL** Copies **all attributes** from the table into the index.
2. **KEYS\_ONLY** Only the primary key attributes (partition + sort keys) are in the index.  
    → Super lightweight
3. **INCLUDE** Copies a specific subset of attributes you define  
    → Balanced between ALL and KEYS\_ONLY

**Example Scenario**

Table: Books (BookID, Title, Author, YearPublished, Genre)

GSI: Index on Author

* With INCLUDE Projection:  
  Only Author + Title are stored in the GSI → Less storage, faster queries when you search books by author and want only titles.

### ***Task 08: How can you say that DYnamoDB prevents data loss?***

**Data Replication**

DynamoDB automatically replicates your data across **multiple Availability Zones (AZs)** in the same AWS Region   
 → So if one AZ goes down (hardware failure, power issue), your data stays safe in other AZs

**Built-In Durability**

AWS promises **99.999999999% (11 9’s) durability** for your data  
That’s insane reliability compared to spinning your own database

**Automatic Backups**

• You can enable **Point-in-Time Recovery (PITR)** → Keeps continuous backups, letting you roll back to any second in the last 35 days   
 • You can also take manual backups on demand

**Strong Consistency Option**

When you need super-strict data consistency, DynamoDB lets you read in **strong consistency mode** → Ensures you always get the latest write

**No Manual Replication Hassles**

* Unlike DIY DB setups where replication config is painful, DynamoDB manages all of that in the background
* No need to set up replication groups, etc.

### ***Task 09: What does in-place atomic updates mean? Does DynamoDB support it?***

Yo, let’s break this down super clean and simple 😎

An **in-place atomic update** means modifying an item in the database directly, without needing to read it first, change it in your app, and write it back.

Atomic → Either the update happens completely or not at all  
In-place → Happens directly on the item in the database

**Example of Atomic Update**

Let’s say you have an item with an attribute:  
Likes = 10

If multiple clients try to increment Likes at the same time, an atomic update ensures that the final value is correct (e.g., no race conditions messing it up).

DynamoDB totally supports in-place atomic updates

It provides an **UpdateItem API** with an **UpdateExpression** like this:

aws dynamodb update-item \

--table-name MyTable \

--key '{"ID": {"N": "1"}}' \

--update-expression "SET Likes = Likes + :inc" \

--expression-attribute-values '{":inc": {"N": "1"}}'

This increments the Likes attribute atomically – no need to read before writing!

### ***Task 10: What are Streams in DynamoDB?***

**DynamoDB Streams** are like a “timeline” of all changes happening in a table.  
Every time an item is **added, updated, or deleted**, DynamoDB can capture that change in a stream.

You can then **process these changes in real-time** using AWS services like Lambda.

**Key Points**

1. **Records Changes**
   * Streams capture INSERT, MODIFY, and REMOVE events.
2. **Retention**
   * Stream records are stored for **24 hours** by default.
3. **Use Cases**
   * Trigger Lambda functions on data changes
   * Sync data with other databases or search engines
   * Maintain materialized views or caches
4. **Stream View Types** You can choose what each record contains:  
   * **KEYS\_ONLY** → Only primary key attributes
   * **NEW\_IMAGE** → The new version of the item after the change
   * **OLD\_IMAGE** → The old version before the change
   * **NEW\_AND\_OLD\_IMAGES** → Both old and new versions

**Example Scenario**

Table: Orders

* You attach a DynamoDB Stream → Whenever a new order is placed (INSERT)
* A Lambda function is triggered → Sends an email confirmation automatically

**DynamoDB Streams = real-time change logs for your table**

* Capture inserts, updates, deletes
* Process with Lambda or other services
* Useful for notifications, analytics, replication, caching

### ***Task 11: What are Dynamo DB Pricing Tiers?***

DynamoDB mainly charges you for **reads, writes, storage, and optional features**. There are two main **capacity modes**, which are essentially the “pricing tiers”:

**Provisioned Capacity (Fixed Tier)**

* You **predefine** how many reads and writes per second your table will handle.
* You pay for the **capacity units**, not actual usage.

**Costs:**

* **Read Capacity Units (RCU)** – 1 RCU = 1 strongly consistent read per second for 4 KB of data
* **Write Capacity Units (WCU)** – 1 WCU = 1 write per second for 1 KB of data

**On-Demand Capacity (Pay-per-Request Tier)**

* No need to predefine capacity
* You pay **per read/write request**
* Ideal for **unpredictable or spiky workloads**

**Optional Features (Extra Costs)**

* **DynamoDB Streams** → Charged per read request on the stream
* **Global Tables** → Each replica table incurs read/write costs
* **Backups & PITR (Point-in-Time Recovery)** → Charged per GB per month
* **Data Transfer** → Outbound traffic costs extra

### ***Task 12: Do you have any max limit for Item Size in Dynamo DB?***

Yes, DynamoDB does have a max item size limit

**Maximum Item Size**

* **Single item (including all attributes)** → **400 KB**
* This includes:  
  + Attribute names
  + Attribute values
  + All metadata stored with the item

**Implications**

* You **cannot store huge blobs** (like big images or videos) directly in a single item
* For large objects, the usual pattern is:
  + Store the object in **S3**
  + Keep only the reference (URL/key) in DynamoDB

### ***Task 13: How many GSI’s Global Secondary Indexes can you create in a table?***

**Max GSIs per DynamoDB Table**

* **Maximum Global Secondary Indexes (GSI) per table:** **20**

**Notes**

* Each GSI **incurs additional cost** (read/write units + storage)
* You should **only create the GSIs you really need**, otherwise it gets expensive and can affect performance

**Max 20 GSIs per table**

### ***Task 14: What is Dynamo DB Accelerator?***

**DynamoDB Accelerator (DAX)** is an **in-memory caching service** for DynamoDB that makes reads **super fast** – like microsecond-level fast

* Think of it as a **managed, write-through cache** for DynamoDB
* Fully managed by AWS → no servers to maintain

1. **Speeds Up Reads**
   * Normal DynamoDB reads → milliseconds
   * With DAX → **microseconds**
2. **Fully Compatible**
   * Works with existing DynamoDB tables
   * Minimal code changes → just point your app to DAX cluster
3. **Write-Through Cache**
   * Writes go to both DynamoDB and the cache automatically
   * Ensures data consistency
4. **Use Cases**
   * Real-time applications needing ultra-low latency
   * Gaming leaderboards, high-traffic mobile apps, live dashboards

**DAX = DynamoDB + super fast in-memory cache**

* Microsecond reads
* Fully managed
* Automatic cache updates
* Ideal for read-heavy workloads

### ***Task 15: What are Dynamo DB Global tables?***

**Global Tables** let you have a **single DynamoDB table replicated automatically across multiple AWS regions**.

* Changes made in one region **propagate to all other regions** automatically
* Enables **multi-region, fully active-active applications**

**Key Points**

1. **Multi-Region Replication**
   * Your table exists in multiple regions
   * Reads and writes can happen in any region → low-latency global access
2. **High Availability & Disaster Recovery**
   * If one region goes down, your app can still read/write from another region
3. **Fully Managed**
   * AWS handles replication, conflict resolution, and scaling
   * No manual setup for cross-region syncing
4. **Use Cases**
   * Global e-commerce catalogs
   * Gaming apps with worldwide players
   * Real-time collaboration apps

**DynamoDB Global Tables = multi-region replicated table**

* Read/write in multiple regions
* Automatic replication & conflict resolution
* Great for low-latency, highly available global apps

### ***Task 16: What are indexes and Secondary indexes in Dynamo DB?***

**Indexes in DynamoDB**

* An **index** is a way to **query data efficiently** without scanning the whole table.
* Think of it like the **index in a book**: lets you jump directly to what you need instead of reading everything.

**Primary Index**

* The main table’s keys:
  + **Partition Key** (hash key)
  + **Sort Key** (range key, optional)
* Used for normal GetItem and Query operations.

**Secondary Indexes**

* **Optional additional indexes** you create to **query the table using different keys**.
* Two types:

a) Global Secondary Index (GSI)

* Can have **different partition and sort keys** than the main table
* Can span all items in the table
* Useful for queries on attributes not in the primary key

b) Local Secondary Index (LSI)

* Shares the **same partition key** as the main table
* Can have a **different sort key**
* Useful for alternative sort orders of items with the same partition key

**Why Indexes Are Useful**

* Make queries faster → no full table scan
* Support multiple query patterns without duplicating data

### ***Task 17: What are Hot Keys and Hot Partitions?***

**Hot Key**

* A **hot key** is a **partition key that gets too many requests** compared to others.
* Example: You have a table Orders with CustomerID as the partition key.
  + If 90% of requests are for CustomerID = 123, that key is “hot.”
* Problems: Throttling, slower reads/writes for that key

**Hot Partition**

* DynamoDB stores items in **partitions** internally.
* A **hot partition** happens when **most of your traffic hits one partition**.
* Usually caused by bad partition key design (like sequential IDs or heavily skewed values)
* Leads to:
  + Throttled requests
  + Uneven load distribution
  + Higher latency

**How to Avoid Hot Keys / Hot Partitions**

1. **Choose good partition keys**
   * Make keys **evenly distributed** across your dataset
   * Example: Use UUIDs, hashed values, or composite keys
2. **Use Randomization or Sharding**
   * Add a prefix/suffix to the key to spread traffic
3. **Avoid sequential keys**
   * Timestamps or auto-increment IDs → can cause hot partitions

### ***Task 18: What are Table level operations and Item level operations in Dynamo DB?***

**Table-Level Operations**

These are operations that **affect the whole table**.

**Examples:**

* **CreateTable** → Create a new table
* **DeleteTable** → Delete a table
* **UpdateTable** → Change table settings (e.g., provisioned throughput, GSIs)
* **DescribeTable** → Get metadata about the table
* **ListTables** → List all tables

**Item-Level Operations**

These are operations that **affect individual items (rows) in a table**.

**Examples:**

* **PutItem** → Add a new item
* **GetItem** → Retrieve an item by primary key
* **UpdateItem** → Modify attributes of an existing item
* **DeleteItem** → Remove an item
* **BatchGetItem / BatchWriteItem** → Operate on multiple items at once