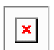
Amazon DynamoDB

Getting Started Guide

API Version 2012-08-10



Amazon DynamoDB: Getting Started Guide

Copyright © 2015 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

The following are trademarks of Amazon Web Services, Inc.: Amazon, Amazon Web Services Design, AWS, Amazon CloudFront, AWS CloudTrail, AWS CodeDeploy, Amazon Cognito, Amazon DevPay, DynamoDB, ElastiCache, Amazon EC2, Amazon Elastic Compute Cloud, Amazon Glacier, Amazon Kinesis, Kindle, Kindle Fire, AWS Marketplace Design, Mechanical Turk, Amazon Redshift, Amazon Route 53, Amazon S3, Amazon VPC, and Amazon WorkDocs. In addition, Amazon.com graphics, logos, page headers, button icons, scripts, and service names are trademarks, or trade dress of Amazon in the U.S. and/or other countries. Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon.

All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Getting Started with Amazon DynamoDB 1
Introduction to DynamoDB Concepts 1
Tables 1
Items 2
Attributes 2
Primary Key 2
Secondary Indexes 2
Next Step 2
Tutorial: Using DynamoDB Local 3
Prerequisites 4
Download and Run DynamoDB Local 4
About the Music Table 4
Next Step 5
Step 1: Create a Table 5
Next Step 6
Step 2: Get Information About Tables 6
Step 2.1: Retrieve a Table Description 7
Step 2.2: Retrieve a List of Your Tables 7
Next Step 8
Step 3: Write Items to the Table 8
Step 3.1: Write a Single Item 8
Step 3.3: Write Multiple Items 10
Next Step 11
Step 4: Read an Item Using Its Primary Key 11
Step 4.1: Read an Item Using GetItem 12
Step 4.2: Retrieve a Subset of Attributes Using a Projection Expression 13
Step 4.4: Read Multiple Items Using BatchGetItem 15
Next Step 16
Step 5: Query and Scan the Table 16
Step 5.1: Run a Query 16
Step 5.2: Filter Query Results 17
Step 5.3: Scan the Table 18
Next Step 18
Step 6: Work with a Secondary Index 18
Step 6.1: Create a Global Secondary Index 19
Step 6.2: Query the Index 20
Step 6.3: Scan the Index 21
Next Step 22
Step 7: Modify Items in the Table 22
Step 7.1: Update an Item 22
Step 7.2: Delete an Item 25
Next Step 26
Step 8: Clean Up 26
Next Step 27
Summary 27
Next Steps 27
Tutorial: Using the AWS SDK for Java 28
Prerequisites 28
Step 1: Create a Table 29
Step 2: Load Sample Data 30
Step 2.1: Download the Sample Data File 31
Step 2.2: Load the Sample Data Into the Movies Table 31
Step 3: Put, Update, and Delete an Item 33
Step 3.1: Add a New Item (PutItem) 33
Step 3.2: Add an Item, Specify Condition (PutItem) 34

Step 3.3: Update the Item (UpdateItem) ... 35
Step 3.4: Increment an Atomic Counter (UpdateItem) ... 37
Step 3.5: Update Item with a Condition (UpdateItem) ... 38
Step 3.6: Delete an Item (DeleteItem) ... 39
Step 4: Query and Scan the Data ... 41
Step 4.1: Query ... 41
Step 4.2: Scan ... 43
Step 5: (Optional) Delete the Table ... 44
Summary ... 45
Using the Amazon DynamoDB Service ... 45
Next Steps ... 53

Getting Started with Amazon DynamoDB

Welcome to the Amazon DynamoDB Getting Started Guide. This guide contains hands-on tutorials to help you learn about Amazon DynamoDB. We strongly encourage you to review this guide in this order:

- Become familiar with the basic DynamoDB concepts on this page.
- Work through the Tutorial: Using DynamoDB Local (p. 3) section. This tutorial uses *DynamoDB Local*, a standalone client-side program that lets you learn about the DynamoDB API at no cost, and with minimal setup and configuration.
- If you are a Java programmer, work through the Tutorial: Using the AWS SDK for Java (p. 28) section. This show you how to write Java programs that you can run against either DynamoDB Local or the Amazon DynamoDB web service.

After you complete the exercises in this guide, we recommend that you read the Amazon DynamoDB Developer Guide. The Amazon DynamoDB Developer Guide provides more in-depth information about DynamoDB, including sample code and best practices.

Topics

- Introduction to DynamoDB Concepts (p. 1)
- Next Step (p. 2)

Introduction to DynamoDB Concepts

This section briefly introduces some of the basic DynamoDB concepts. This will help you as you follow steps in the tutorials.

Tables

Similar to other database management systems, DynamoDB stores data in tables. A *table* is a collection of data. For example, we could create a table named `People`, where we could store information about friends, family, or anyone else of interest. We could also have a `Cars` table where we store information about vehicles that people drive.

Items

Each table contains multiple items. An *item* is a group of attributes that is uniquely identifiable among all of the other items. In a `People` table, each item would represent one person. For a `Cars` table, each item would represent one vehicle. Items are similar in many ways to rows, records, or tuples in relational database systems. In DynamoDB, there is no limit to the number of items that you can store in a table.

Attributes

Each item is composed of one or more attributes. An *attribute* is a fundamental data element, something that does not need to be broken down any further. A *Department* item might have attributes such as `DepartmentID`, `Name`, `Manager`, and so on. An item in an `People` table could contain attributes such as `PersonID`, `LastName`, `FirstName`, and so on. Attributes in DynamoDB are similar in many ways to fields or columns in other database management systems.

Primary Key

When you create a table, in addition to the table name, you must specify the primary key of the table. As in other databases, a primary key in DynamoDB uniquely identifies each item in the table, so that no two items can have the same key. When you add, update, or delete an item in the table, you must specify the primary key attribute values for that item. The key values are required; you cannot omit them.

DynamoDB supports two different kinds of primary keys:

- Primary key consisting of a hash attribute—The primary key consists of a single attribute, which is called a *hash attribute*. Each item in the table is uniquely identified by its hash attribute value.
- Primary key consisting of hash and range attributes—The primary key consists of two attributes. The first attribute is the hash attribute and the second one is the *range attribute*. Each item in the table is uniquely identified by the combination of its hash and range attribute values. It is possible for two items to have the same hash attribute value, but those two items must have different range attribute values.

Secondary Indexes

In DynamoDB, you can read data in a table by providing primary key attribute values. If you want to read the data using non-key attributes, you can do so using a secondary index. After you create a secondary index on a table, you can read data from the index in much the same way as you do from the table. By using secondary indexes, your applications can use many different query patterns, in addition to access the data by primary key values.

For more information, see the following topics in the Amazon DynamoDB Developer Guide:

- Data Model Concepts - Tables, Items, and Attributes
- Primary Key
- Secondary Indexes

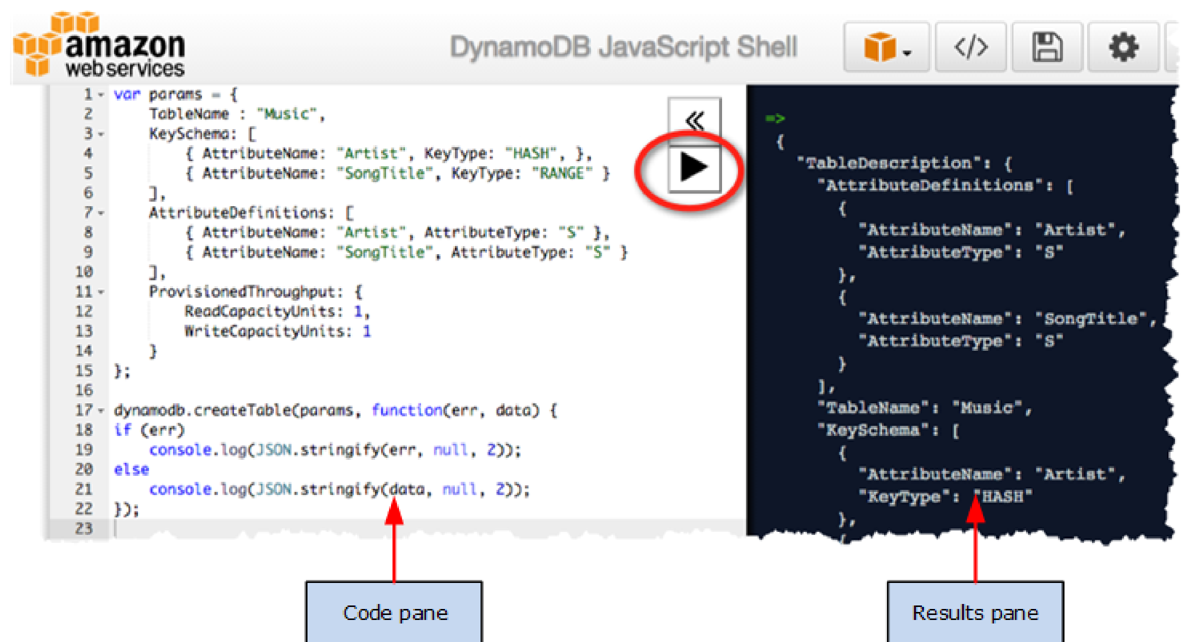
Next Step

Tutorial: Using DynamoDB Local (p. 3)

Tutorial: Using DynamoDB Local

In this tutorial, you will learn the basics of DynamoDB operations. You will do this using DynamoDB Local, a program that provides a DynamoDB-compatible database that runs locally on your computer. Using DynamoDB Local provides the following:

- DynamoDB Local is free of charge, and it does not use DynamoDB service. You can use DynamoDB Local for application development and testing, and then run that same application against the DynamoDB service.
- It provides a JavaScript shell, an interactive environment, for you to try the DynamoDB API:



You enter JavaScript code on the left side, and run the code. The right side shows you the results.

This tutorial provides JavaScript code snippets that you copy and paste into the JavaScript shell.

In this exercise you will create a table called `Music` and perform various operations on it, including add items, modify items, and read items.

At the end of this tutorial, you will have gained hands-on experience with most of the DynamoDB API operations.

Cost: Free, because you use DynamoDB Local rather than the Amazon DynamoDB service.

Prerequisites

Before you begin this tutorial, you need to download and run DynamoDB Local. After that, you will be able to access the built-in JavaScript shell.

Download and Run DynamoDB Local

DynamoDB Local is available as an executable `.jar` file. It will run on Windows, Linux, Mac OS X, and other platforms that support Java. Follow the steps in this procedure to download and run DynamoDB Local.

1. Download DynamoDB Local for free using one of these links:
   - *.tar.gz* format: http://dynamodb-local.s3-website-us-west-2.amazonaws.com/dynamodb_local_latest.tar.gz
   - *.zip* format: http://dynamodb-local.s3-website-us-west-2.amazonaws.com/dynamodb_local_latest.zip

   Important
   DynamoDB Local supports the Java Runtime Engine (JRE) version 6.x or newer; it will not run on older JRE versions.
2. After you have downloaded the archive to your computer, extract the contents and copy the extracted directory to a location of your choice.
3. To start DynamoDB Local, open a command prompt window, navigate to the directory where you extracted `DynamoDBLocal.jar`, and enter the following command:

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -
sharedDb
```

   Note
   DynamoDB Local uses port 8000 by default. If port 8000 is unavailable, this command will throw an exception. You can use the `-port` option to specify a different port number. For a complete list of DynamoDB Local runtime options, including `-port` , type this command: `java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -help`
   If you need to stop DynamoDB Local, you can do so by pressing `Ctrl-C`.
4. You can now access the built-in JavaScript shell.

   Important
   We recommend that you run the DynamoDB Local JavaScript shell on Firefox or Chrome. If you run the JavaScript shell in other browsers, errors may occur.

   Open a web browser on your computer and go to the following URL: http://localhost:8000/shell

About the Music Table

The primary key is made of artist (hash attribute), and song title (range attribute). When you create the table, you only need to identify primary key attributes and their data types. Other than the required primary

key, a DynamoDB table does not require a schema. Each item in a DynamoDB table can contain a varying number of different attributes and attribute types. For illustration, we will store attributes such as `AlbumTitle` (string type), `Year`, `Price` (number type), `Composers` (list type), and `Tags` (map type, an unordered collection of name/value pairs).

Following is a sample item:

```
Item: {
    "Artist":"No One You Know",
    "SongTitle":"Call Me Today",
    "AlbumTitle":"Somewhat Famous",
    "Year": 2015,
    "Price": 2.14,
    "Genre": "Country",
    "Tags": {
        "Composers": [
              "Smith",
              "Jones",
              "Davis"
        ],
        "LengthInSeconds": 214
    }
}
```

Next Step

- Step 1: Create a Table (p. 5)

Step 1: Create a Table

In this step, you will create a table named *Music*. You will use the `CreateTable` API operation to do this. The primary key for the table will consist of two attributes that are both string type: `Artist` (hash key) and `SongTitle` (range key).

1. Copy the following code and paste it into the left side of the DynamoDB Local JavaScript shell window.

```
var params = {
    TableName : "Music",
    KeySchema: [
        { AttributeName: "Artist", KeyType: "HASH", },
        { AttributeName: "SongTitle", KeyType: "RANGE" }
    ],
    AttributeDefinitions: [
        { AttributeName: "Artist", AttributeType: "S" },
        { AttributeName: "SongTitle", AttributeType: "S" }
    ],
    ProvisionedThroughput: {
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1
    }
};

dynamodb.createTable(params, function(err, data) {
```

```
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

In the code, you specify the table name, its primary key attributes and their data types. Provisioned throughput is required as shown but is not applicable when using DynamoDB Local, it is ignored. It is beyond the scope of this exercise but explained later.

2. Click the play button arrow to run the code, as shown in the following screen shot. The response from DynamoDB Local is shown in the right side of the window.

In the response, take note of the `TableStatus`. Its value should be `ACTIVE`. This indicates that the *Music* table is ready for use.

In the code snippet, note the following:

- The `params` object holds the parameters for the corresponding DynamoDB API operation.
- The `dynamodb.`*`<operation>`* line invokes the operation, with the correct parameters. In the example above, the operation is `createTable`.

Next Step

Step 2: Get Information About Tables (p. 6)

Step 2: Get Information About Tables

DynamoDB stores detailed metadata about your tables, such as table name, its primary key attributes, table status, and provisioned throughput settings. In this section you will retrieve information about the

music table using the DynamoDB `DescribeTable` operation and also obtain a list of tables using the `ListTables` operation.

Topics

- Step 2.1: Retrieve a Table Description (p. 7)
- Step 2.2: Retrieve a List of Your Tables (p. 7)
- Next Step (p. 8)

Step 2.1: Retrieve a Table Description

Use the DynamoDB `DescribeTable` operation to view details about a table.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music"
};

dynamodb.describeTable(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. The response from DynamoDB Local contains a complete description of the table.

Step 2.2: Retrieve a List of Your Tables

Use the `ListTables` API operation to list the names of all of your tables. This operation does not require any parameters.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {};

dynamodb.listTables(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. The response from DynamoDB Local contains just one table called `Music`.

Next Step

Step 3: Write Items to the Table (p. 8)

Step 3: Write Items to the Table

When you write an item to a DynamoDB table, only the primary key attribute(s) are required. Other than the primary key, the table does not require a schema. In this section, you will write an item to a table (`PutItem` operation), write an item conditionally, and also write multiple items in a single operation (`BatchWriteItem` operation).

Topics

- Step 3.1: Write a Single Item (p. 8)
- Step 3.3: Write Multiple Items (p. 10)
- Next Step (p. 11)

Step 3.1: Write a Single Item

Use the `PutItem` API operation to write an item.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music",
    Item: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today",
        "AlbumTitle":"Somewhat Famous",
        "Year": 2015,
        "Price": 2.14,
        "Genre": "Country",
        "Tags": {
            "Composers": [
                  "Smith",
                  "Jones",
                  "Davis"
            ],
            "LengthInSeconds": 214
        }
    }
};

dynamodb.putItem(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. If the write is successful, the response will be an empty map: {}

Note the following about the item you just added:

- `Artist` and `SongTitle` are *primary key attributes* (hash and range attributes respectively). Both are of string type. Every item that you add to the table must have values for these attributes.
- Other attributes are `AlbumTitle` (string), `Year` (number), `Price` (number), `Genre` (string), and `Tags` (map).
- DynamoDB allows you to *nest* attributes within other attributes. The `Tags` map contains two *nested attributes*—`Composers` (list) and `LengthInSeconds` (number).
- `Artist`, `SongTitle`, `AlbumTitle`, `Year`, `Price`, `Genre`, and `Tags` are *top-level attributes*, because they are not nested within any other attributes.

For more information, see Data Types in the Amazon DynamoDB Developer Guide.

Step 3.2: Perform a Conditional Write

By default, `PutItem` does not check first to see if there is already an item with the same key, it simply overwrites any existing item. If you want to ensure that you do not overwrite an existing item, you can add a `ConditionExpression` parameter. This is a logical condition that must be satisfied in order for the write to succeed.

In this step you will try to write the same item, but this time you will specify a condition to see whether an item with the same primary key already exists. The write will fail, because there is already an item in the table with the same primary key.

1. Modify the `params` object so that it looks like this:

```
var params = {
    TableName: "Music",
    Item: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today",
        "AlbumTitle":"Somewhat Famous",
        "Year": 2015,
        "Price": 2.14,
        "Genre": "Country",
        "Tags": {
            "Composers": [
                  "Smith",
                  "Jones",
                  "Davis"
            ],
            "LengthInSeconds": 214
        }
    },
   "ConditionExpression": "attribute_not_exists(Artist) and attribute_not_ex
ists(SongTitle)"
};
```

Note

The only difference is the `ConditionExpression` parameter. This will prevent you from overwriting the item. if there is already an item in the table with the same primary key values (`No One You Know`, `Call Me Today`).

For more information, see Condition Expressions in the Amazon DynamoDB Developer Guide.

2. Click the play button arrow to run the code.

   The conditional write fails because the item already exists.

Step 3.3: Write Multiple Items

You can use the `BatchWriteItem` operation to perform multiple writes in one step. The following code adds several items, with the required primary key attributes, but different non-key attributes.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    RequestItems: {
        "Music": [
            {
                PutRequest: {
                    Item: {
                        "Artist": "No One You Know",
                        "SongTitle": "My Dog Spot",
                        "AlbumTitle":"Hey Now",
                        "Price": 1.98,
                        "Genre": "Country",
                        "CriticRating": 8.4
                    }
                }
            },
            {
                PutRequest: {
                    Item: {
                        "Artist": "No One You Know",
                        "SongTitle": "Somewhere Down The Road",
                        "AlbumTitle":"Somewhat Famous",
                        "Genre": "Country",
                        "CriticRating": 8.4,
                        "Year": 1984
                    }
                }
            },
            {
                PutRequest: {
                    Item: {
                        "Artist": "The Acme Band",
                        "SongTitle": "Still In Love",
                        "AlbumTitle":"The Buck Starts Here",
                        "Price": 2.47,
                        "Genre": "Rock",
                        "PromotionInfo": {
                            "RadioStationsPlaying":[
```

```
                                "KHCR", "KBQX", "WTNR", "WJJH"
                            ],
                            "TourDates": {
                                "Seattle": "20150625",
                                "Cleveland": "20150630"
                            },
                            "Rotation": "Heavy"
                        }
                    }
                }
            },
            {
                PutRequest: {
                    Item: {
                        "Artist": "The Acme Band",
                        "SongTitle": "Look Out, World",
                        "AlbumTitle":"The Buck Starts Here",
                        "Price": 0.99,
                        "Genre": "Rock"
                    }
                }
            }
        ]
    }
};

dynamodb.batchWriteItem(params, function (err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. If the batch write is successful, the response will contain the following: `"UnprocessedItems": {}`. This indicates that all of the items in the batch have been written.

Next Step

Step 4: Read an Item Using Its Primary Key (p. 11)

Step 4: Read an Item Using Its Primary Key

DynamoDB provides the `GetItem` operation for retrieving one item at a time. You can retrieve an entire item, or a subset of its attributes.

DynamoDB supports the Map and List attribute types. These attribute types allow you to nest other attributes within them, so that you can store complex documents in an item. You can use `GetItem` to retrieve an entire document, or just some of the nested attributes within that document.

Step 4.1: Read an Item Using GetItem

Use the `GetItem` API operation to read an item. You must provide the primary key of the item you want. DynamoDB will then read the item directly from its physical location in the database.

The following code reads an item from the `Music` table by specifying the primary key.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    }
};

dynamodb.getItem(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. The requested item appears in the response.
3. (Optional) You can change the primary key in the preceding code to retrieve different music items. Let us retrieve different music items from the table.

   Modify the `params` object, using any of the primary key values shown below. Click the play button arrow to run the code, and verify that the correct item is returned in the response.

```
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "My Dog Spot"
    }
```

```
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Somewhere Down The Road"
    }
```

```
    Key: {
        "Artist": "The Acme Band",
        "SongTitle": "Still In Love"
    }
```

```
    Key: {
        "Artist": "The Acme Band",
        "SongTitle": "Look Out, World"
    }
```

Step 4.2: Retrieve a Subset of Attributes Using a Projection Expression

By default, the `GetItem` API operation returns all of the attributes in the item. To return only some of the attributes, you provide a *projection expression* — a comma-separated string of attribute names that you want to use.

1. In the DynamoDB Local JavaScript shell window, modify the `params` object so that it looks like this:

```
var params = {
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    },
    ProjectionExpression: "AlbumTitle"
};
```

2. Click the play button arrow to run the code. Only one attribute (`AlbumTitle`) appears in the response.

Handling Attribute Names that Are Also Reserved Words

In DynamoDB, you have a great deal of flexibility when it comes to naming your tables and attributes. However, it is possible that a name you choose might conflict with a reserved word. In this situation, you can define an *expression attribute name* and use it in the projection expression.

For the complete list, see Reserved Words in the Amazon DynamoDB Developer Guide.

1. Modify the projection expression so that it also includes the `Year` attribute, which is a reserved word and therefore the `GetItem` operation will fail:

```
var params = {
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    },
    ProjectionExpression: "AlbumTitle, Year"
};
```

2. Click the play button arrow to run the code. An error message appears in the response:

```
Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved
keyword: Year
```

3. Modify the `params` object to use a placeholder token (#y) in `ProjectionExpression`, and then define the placeholder in the `ExpressionAttributeNames` parameter.

```
var params = {
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    },
    ProjectionExpression: "AlbumTitle, #y",
    ExpressionAttributeNames: {"#y": "Year"}
};
```

   In the `ProjectionExpression`, the word `Year` is replaced by the token #y. The `#` (pound sign) is required, and indicates that this is a placeholder. The `ExpressionAttributeNames` parameter indicates that `#y` is to be replaced by `Year` at runtime.

4. Click the play button arrow to run the code. The `AlbumTitle` and `Year` attributes appear in the response.

Step 4.3: Retrieve Nested Attributes Using Document Path Notation

DynamoDB supports a map type attribute to store documents. In the `Music` table, we use a map type attribute called `Tags` to store information such as list of music composers, song duration information, and so on. These are nested attributes. You can retrieve entire document, or a subset of these nested attributes by specifying document path notation.

A *document path* tells DynamoDB where to find the attribute, even if it is deeply nested within multiple lists and maps. In a document path, use the following operators to access nested attributes:

- For a list, use square brackets: **[*n*]**, where *n* is the element number. List elements are zero-based, so [0] represents the first element in the list, [1] represents the second, and so on.
- For a map, use a dot: **.** The dot acts as a separator between elements in the map.

1. Modify the `params` object so that it looks like this:

```
var params = {
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    },
    ProjectionExpression: "AlbumTitle, #y, Tags.Composers[0],
Tags.LengthInSeconds",
    ExpressionAttributeNames: {"#y": "Year"}
};
```

2. Click the play button arrow to run the code. The response contains only the top-level and nested attributes that were specified in `ProjectionExpression`.

Step 4.4: Read Multiple Items Using BatchGetItem

The `GetItem` operation retrieves a single item by its primary key. DynamoDB also supports `BatchGetItem` operation for you to read multiple items in a single request. You specify a list of primary keys for this operation.

The following example retrieves a group of music items. The example also specifies the optional `ProjectionExpression` to retrieve only a subset of the attributes.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    RequestItems: {
        "Music": {
            Keys: [
                {
                    "Artist": "No One You Know",
                    "SongTitle": "My Dog Spot"
                },
                {
                    "Artist": "No One You Know",
                    "SongTitle": "Somewhere Down The Road"
                },
                {
                    "Artist": "The Acme Band",
                    "SongTitle": "Still In Love"
                },
                {
                    "Artist": "The Acme Band",
                    "SongTitle": "Look Out, World"
                }
            ],
            ProjectionExpression:"PromotionInfo, CriticRating, Price"
        }
    }
};

dynamodb.batchGetItem(params, function (err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. The response contains all of the attributes specified in *ProjectionExpression*. If one of the items does not have an attribute, it appears in the response as an empty map: { }
3. (Optional) Remove the `ProjectionExpresion` entirely, and retrieve all of the attributes from the items.
4. (Optional) Add a new `ProjectionExpression` that retrieves at least one nested attribute. Use document path notation to do this.

Next Step

Step 5: Query and Scan the Table (p. 16)

Step 5: Query and Scan the Table

DynamoDB supports the `Query` operation on tables whose primary key is composed of both hash and range attributes. You can also filter query results.

In addition, DynamoDB also supports the `Scan` operation on a table. This section provides introductory examples of using these operations.

Topics

- Step 5.1: Run a Query (p. 16)
- Step 5.2: Filter Query Results (p. 17)
- Step 5.3: Scan the Table (p. 18)
- Next Step (p. 18)

Step 5.1: Run a Query

This section provides examples of Query operations. The queries are specified against the `Music` table. Remember, the table primary key is made of `Artist` (hash attribute) and `Title` (range attributes).

- Query using only the hash attribute of the primary key. For example, find songs by an artist.
- Query using both hash and range attributes of a primary key. For example, find songs by an artist and song title starting with a specific string.
- Filter query results. Find songs by an artist and then return only those songs that have more than three radio stations playing them.

Query Using a Hash Key Attribute

Follow these steps to query for songs by an artist. Note that you use the `KeyConditionExpression` to specify the primary key. In this example it specifies only the hash attribute (`Artist`).

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music",
    KeyConditionExpression: "Artist = :artist",
    ExpressionAttributeValues: {
        ":artist": "No One You Know"
    }
};

dynamodb.query(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
```

```
});
```

In the KeyConditionExpression, :artist is a token. The token value is provided in the ExpressionAttributeValues parameter. The : (colon) indicates that this is a placeholder for the value.

2. Click the play button arrow to run the code. Only the songs by the band No One You Know are returned.

Query Using Hash and Range Key Attributes

Follow these steps to query for songs by an artist (The Acme Band) with song title starting with a specific string (S). Note that you use the KeyConditionExpression to specify the primary key. In this example it specifies both the hash and range attributes of the primary key.

1. Modify the params object from Query Using a Hash Key Attribute (p. 16) so that it looks like this:

```
var params = {
    TableName: "Music",
    ProjectionExpression: "SongTitle",
    KeyConditionExpression: "Artist = :artist and begins_with(SongTitle,
:letter)",
    ExpressionAttributeValues: {
        ":artist": "The Acme Band",
        ":letter": "S"
    }
};
```

Note the use of ProjectionExpression, which will cause the query to return the SongTitle attribute only.

2. Click the play button arrow to run the code. Only songs by The Acme Band, with titles that begin with the letter S, are returned. (There is only one song that meets this criteria in the Music table.)

Step 5.2: Filter Query Results

You can filter results of a query by adding the FilterExpression parameter. In this example you specify a query to find songs by an artist (The Acme Band). The query also specifies the FilterExpression to request DynamoDB to return only the song items that are being played on more than three radio stations.

1. Modify the params object from Query Using a Hash Key Attribute (p. 16) so that it looks like this:

```
var params = {
    TableName: "Music",
    ProjectionExpression: "SongTitle, PromotionInfo.Rotation",
    KeyConditionExpression: "Artist = :artist",
   FilterExpression: "size(PromotionInfo.RadioStationsPlaying) >= :howmany",

    ExpressionAttributeValues: {
        ":artist": "The Acme Band",
```

```
        ":howmany": 3
    },
};
```

Note the use of `ProjectionExpression`, which will cause the query to return the top-level *SongTitle* attribute and the nested *PromotionInfo.Rotation* attribute.

Also the FilterExpression specifies the size() function. For a list of predefined functions, go to Expression Reference in the Amazon DynamoDB Developer Guide.

2. Click the play button arrow to run the code. The response contains the only song by The Acme Band that is in heavy rotation on at least three radio stations.

Step 5.3: Scan the Table

You can use the `Scan` operation to retrieve all of the items in a table. In the following example you scan the `Music` table.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music"
};

dynamodb.scan(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. All of the table items appear in the response.

Next Step

Step 6: Work with a Secondary Index (p. 18)

Step 6: Work with a Secondary Index

Without an index, you can query for items based on primary key. You can add indexes to your table depending on your query patterns. DynamoDB supports two different kinds of indexes:

- Global secondary index — an index with a hash and range key that can be different from those on the table. You can create or delete a global secondary index on a table at any time.
- Local secondary index — an index that has the same hash key attribute as the primary key of the table, but a different range key. You can only create a local secondary index when you create a table; when you delete the table, the local secondary index is also deleted.

For more information about these indexes, go to Improving Data Access with Secondary Indexes in DynamoDB in the Amazon DynamoDB Developer Guide.

In this step, you will add a secondary index to the *Music* table. You will then query and scan the index, in the same way as you would query or scan a table. In this tutorial, you will create and use a global secondary index.

Topics

- Step 6.1: Create a Global Secondary Index (p. 19)
- Step 6.2: Query the Index (p. 20)
- Step 6.3: Scan the Index (p. 21)
- Next Step (p. 22)

Step 6.1: Create a Global Secondary Index

The `Music` table has a primary key made of `Artist` (hash attribute) and `SongTitle` (range attribute). Now suppose you want to query this table by `Genre` and find all of the `Country` songs. Searching on the primary key will help in this case. To do this, we will build a secondary index with `Genre` as the hash attribute.

To make this interesting, we will add the `Price` attribute as the range key. So you can now run a query to find all `Country` songs with `Price` less than 0.99.

You can add an index at the time that you create a table or later using the `UpdateTable` operation.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music",
    AttributeDefinitions:[
        {AttributeName: "Genre", AttributeType: "S"},
        {AttributeName: "Price", AttributeType: "N"}
    ],
    GlobalSecondaryIndexUpdates: [
        {
            Create: {
                IndexName: "GenreAndPriceIndex",
                KeySchema: [
                    {AttributeName: "Genre", KeyType: "HASH"},
                    {AttributeName: "Price", KeyType: "RANGE"},
                ],
                Projection: {
                    "ProjectionType": "ALL"
                },
                ProvisionedThroughput: {
                    "ReadCapacityUnits": 1,"WriteCapacityUnits": 1
                }
            }
        }
    ]
};

dynamodb.updateTable(params, function(err, data) {
```

```
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

In the code:

- `AttributeDefinitions` lists data types of attributes that are later defined as hash and range attributes of the index.
- `GlobalSecondaryIndexUpdates` specifies the index operations. You can create index, update index, or delete an index.
- `ProvisionedThroughput` is not discussed in this exercise (and not applicable when using DynamoDB Local), however note that it is required when creating global secondary indexes.

2. Click the play button arrow to run the code.

   In the response, take note of the `IndexStatus`. Its value should be `CREATING`, which indicates that the index is being built In DynamoDB Local, the new index should be available for use within a few seconds.

   However, when you create an index in the DynamoDB service, there will be a delay while DynamoDB allocates resources for the index, and then begins copying the data from the table into the index. This process can take a long time with very large tables. During this time, the value for `IndexStatus` is `CREATING`. When the status changes to `ACTIVE`, you can begin using the index.

Step 6.2: Query the Index

Now we will use the index to query for all `Country` songs. The index has all of the data you need, so you query the index and not the table.

You use the same `Query` operation to query a table (see Step 5: Query and Scan the Table (p. 16)) or an index on the table. When you query an index you specify both the table name and the index name.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
    KeyConditionExpression: "Genre = :genre",
    ExpressionAttributeValues: {
        ":genre": "Country"
    },
    ProjectionExpression: "SongTitle, Price"
};

dynamodb.query(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
```

```
});
```

2. Click the play button arrow to run the code. Only the `Country` songs are returned.
3. Now let us query for `Country` songs that cost more than two dollars. Here you specify both the hash and range key values for index. Modify the `params` object so that it looks like this:

```
var params = {
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
    KeyConditionExpression: "Genre = :genre and Price > :price",
    ExpressionAttributeValues: {
        ":genre": "Country",
        ":price": 2.00
    },
    ProjectionExpression: "SongTitle, Price"
};
```

4. Click the play button arrow to run the code. This query uses both of the index key attributes (`Genre` and `Price`), returning only the `Country` songs that cost more than 2.00.

Step 6.3: Scan the Index

You can scan an index (using the `Scan` operation) in the same way that you scan a table. When scanning an index, you provide both the table name and index name.

In this example, we will scan the entire global secondary index you created, but we'll retrieve specific attributes only.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
    ProjectionExpression: "Genre, Price, SongTitle, Artist, AlbumTitle"
};

dynamodb.scan(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. All of the items in the index are returned.
3. (Optional) Note that there are only four items in the index (`Count`), but there are five items in the table. The reason is that one of the items does not have a `Price` attribute, so that item was not included in `GenreAndPriceIndex`.

Which of the songs in `Music` items does not have a `Price` attribute? Can you determine which one it is?

Next Step

Step 7: Modify Items in the Table (p. 22)

Step 7: Modify Items in the Table

You can modify an item in a table using `UpdateItem` or you can delete an item using the `DeleteItem` operations. You can update item by updating values of existing attributes, add new attributes, or remove existing attributes. You can use keywords in the `UpdateItem` operation such as `Set` and `Remove` to request specific updates.

Topics

- Step 7.1: Update an Item (p. 22)
- Step 7.2: Delete an Item (p. 25)
- Next Step (p. 26)

Step 7.1: Update an Item

The `UpdateItem` API operation lets you do the following:

- Add more attributes to an item.
- Modify the values of one or more attributes in the item.
- Remove attributes from the item.

To specify which operations to perform, you use an update expression. An *update expression* is a string containing attribute names, operation keywords (such as `SET` and `REMOVE`), and new attribute values. You can combine multiple updates in a single update expression.

For more information, see Reserved Words in the Amazon DynamoDB Developer Guide.

By default, `UpdateItem` operation does not return any data (empty response). You can optionally specify the `ReturnValues` parameter to request attribute values as they appeared before or after the update:

- `ALL_OLD` returns all attribute values as they appeared before the update.
- `UPDATED_OLD` returns only the updated attributes as they appeared before the update.
- `ALL_NEW` returns all attribute values as they appear after the update.
- `UPDATED_NEW` returns only the updated attributes as they appeared after the update.

In this example, you will perform a couple of updates to an item in the `Music` table.

1. The following example updates a `Music` table item by adding a new `RecordLabel` attribute using the `UpdateExpression` parameter.

   Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET RecordLabel = :label",
    ExpressionAttributeValues: {
        ":label": "Global Records"
    },
    ReturnValues: "ALL_NEW"
};

dynamodb.updateItem(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. Verify that the item in the response has a `RecordLabel` attribute.
3. We will now apply multiple changes to the item using the `UpdateExpression` parameter: Change the price, and remove one of the composers.

   Modify the `params` object so that it looks like this:

```
var params = {
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression:
        "SET Price = :price REMOVE Tags.Composers[2]",
    ExpressionAttributeValues: {
        ":price": 0.89
    },
    ReturnValues: "ALL_NEW"
};
```

4. Click the play button arrow to run the code. Verify that the `UpdateExpression` worked.

Specify a Conditional Write

By default updates are performed unconditionally. You can specify a condition in the `UpdateItem` operation to perform conditional update. For example, you may want to check if an attribute exists before changing its value, or check the existing value and apply an update only if the existing value meets certain criteria.

The `UpdateItem` operation provides `ConditionExpression` parameter for you to specify one or more conditions.

In this example, you add an attribute only if it doesn't already exist.

1. Modify the `params` object from Step 7.1: Update an Item (p. 22) so that it looks like this:

```
var params = {
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET RecordLabel = :label",
    ExpressionAttributeValues: {
        ":label": "New Wave Recordings, Inc."
    },
    ConditionExpression: "attribute_not_exists(RecordLabel)",
    ReturnValues: "ALL_NEW"
};
```

2. Click the play button arrow to run the code. This should fail with response: `The conditional request failed`, because the item already has the `RecordLabel` attribute.

Specify an Atomic Counter

DynamoDB supports atomic counters, where you use the `UpdateItem` operation to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they were received.) For example, a music player application might want to maintain a counter each time song is played. In this case, the application would need to increment this counter regardless of its current value. For more information, go to Atomic Counters in the Amazon DynamoDB Developer Guide.

In this example we will first use the `UpdateItem` operation to add an attribute (`Plays`) to keep track of the number of times the song is played. Then, using another `UpdateItem` operation, we will increment its value by 1.

1. Modify the `params` object from Step 7.1: Update an Item (p. 22) so that it looks like this:

```
var params = {
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET Plays = :val",
    ExpressionAttributeValues: {
        ":val": 0
    },
    ReturnValues: "UPDATED_NEW"
};
```

2. Click the play button arrow to run the code. The `Plays` attribute is added, and its value (zero) is shown in the response.
3. Now modify the `params` object so that it looks like this:

```
var params = {
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET Plays = Plays + :incr",
    ExpressionAttributeValues: {
        ":incr": 1
    },
    ReturnValues: "UPDATED_NEW"
};
```

4. Click the play button arrow to run the code. The `Plays` attribute is incremented by one.
5. Run the code a few more times. Each time you do this, `Plays` is incremented.

Step 7.2: Delete an Item

You will now use the `DeleteItem` API operation to delete an item from the table. Note that this operation is permanent—there is no way to restore an item.

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music",
    Key: {
        Artist: "The Acme Band",
        SongTitle: "Look Out, World"
    }
};

dynamodb.deleteItem(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code. The item is deleted.

Specify a Conditional Delete

By default, a delete operation is unconditional. However, you can use the `ConditionExpression` parameter to perform conditional deletes. In this example you delete an item, only if its price is 0.

1. Modify the `params` object so that it looks like this:

```
var params = {
    TableName: "Music",
    Key: {
        Artist: "No One You Know",
        SongTitle: "My Dog Spot"
    },
    ConditionExpression: "Price = :price",
    ExpressionAttributeValues: {
        ":price": 0.00
    }
};
```

2. Click the play button arrow to run the code. The conditional delete fails because the song is not free (`Price` is not 0.00).

Next Step

Step 8: Clean Up (p. 26)

Step 8: Clean Up

In this step, you will use the `DeleteTable` API operation to remove the table. When you do this, the `Music` table, `GenreAndPriceIndex`, and all of the data will be permanently deleted. This operation cannot be undone.

To delete the table

1. Replace everything in the left side of the DynamoDB Local JavaScript shell window with the following code:

```
var params = {
    TableName: "Music"
};

dynamodb.deleteTable(params, function(err, data) {
    if (err)
        console.log(JSON.stringify(err, null, 2));
    else
        console.log(JSON.stringify(data, null, 2));
});
```

2. Click the play button arrow to run the code to delete the table.

In DynamoDB Local, tables are deleted immediately.

However, if you delete a table in the DynamoDB service, there will be a delay while DynamoDB releases the resources that the table was using. If you issue a `DescribeTable` request during this time, the value for `TableStatus` will be `DELETING`. When the table resources are released, the table no longer exists, so a subsequent `DescribeTable` request will result in an error message.

Next Step

Summary (p. 27)

Summary

In this exercise you used DynamoDB Local to learn about various DynamoDB operations. Using the built-in JavaScript shell you were able to run JavaScript snippets provided. We used the shell because you could easily copy/paste the code and run it.

All this is done on your local computer. This did not use the DynamoDB service. So the DynamoDB Local provides an ideal development environment for you to build and test your application before running it again the DynamoDB service.

To learn more about these DynamoDB operations see the following topics in the Amazon DynamoDB Developer Guide:

- Working With Tables
- Working With Items
- Query and Scan
- Secondary Indexes
- Best Practices

Next Steps

For an additional getting started exercise, see the following section of this guide:

- Tutorial: Using the AWS SDK for Java (p. 28)

Tutorial: Using the AWS SDK for Java

In this tutorial, you will use the AWS SDK for Java to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

The SDK for Java offers several programming models for different use cases. In this exercise, the Java code uses the document model that provides a level of abstraction that makes it easier for you to work with JSON documents.

You will use DynamoDB Local in this tutorial. In the Summary (p. 45), we will explain how to run the same code against the DynamoDB service.

Cost: Free

Prerequisites

Before you begin this tutorial, you need to do the following:

- Download and run DynamoDB Local. For more information, see Download and Run DynamoDB Local (p. 4).
- Sign up for Amazon Web Services. You need security credentials to use AWS SDKs, which you get when you sign up for AWS. For sign up instructions, go to http://aws.amazon.com.
- Setup the AWS SDK for Java. You need to set up the following:
  - Install a Java development environment. If you are using Eclipse IDE, install the AWS Toolkit for Eclipse.
  - Install the AWS SDK for Java.
  - Setup your AWS security credentials for use with the SDK for Java.

  For instructions, see Getting Started in the AWS SDK for Java Developer Guide.

Tip
As you work through this tutorial, you can refer to the SDK for Java documentation in Javadoc format. The Javadocs are available at http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/.

Next Step

Step 1: Create a Table (p. 29)

Step 1: Create a Table

In this step, you will create a table named `Movies`. The primary key for the table is composed of the following hash and range attributes:

- `year`—the hash attribute of number type.
- `title`—the range attribute of string type.

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import java.util.Arrays;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;

public class MoviesCreateTable {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        String tableName = "Movies";
        Table table = dynamoDB.createTable(tableName,
                Arrays.asList(
                        new KeySchemaElement("year", KeyType.HASH),
                        new KeySchemaElement("title", KeyType.RANGE)),
                Arrays.asList(
                        new AttributeDefinition("year", ScalarAttribute
Type.N),
                        new AttributeDefinition("title", ScalarAttribute
Type.S)),
                new ProvisionedThroughput(10L, 10L));
```

```
        System.out.println("Table status: " + table.getDescription().get
TableStatus());
    }
}
```

Note

- You set the endpoint (`client.setEndpoint`) to indicate that you are creating the table in DynamoDB Local.
- In the `createTable` call, you specify table name, primary key attributes, and its data types.
- Provisioned throughput is required as shown, but it does not apply when using DynamoDB Local (it is ignored). Provisioned throughput is beyond the scope of this exercise, but it is explained later in this tutorial.

2. Compile and run the program.

Next Step

Step 2: Load Sample Data (p. 30)

Step 2: Load Sample Data

In this step, you will populate the `Movies` table with sample data.

Topics

- Step 2.1: Download the Sample Data File (p. 31)
- Step 2.2: Load the Sample Data Into the Movies Table (p. 31)

We will use a sample data file containing information from a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[
   {
      "year" : ... ,
      "title" : ... ,
      "info" : { ... }
   },
   {
      "year" : ...,
      "title" : ...,
      "info" : { ... }
   },

    ...

]
```

In the JSON data, note the following:

- We use the `year` and `title` as the primary key attribute values for our `Movies` table.
- We store the rest of the `info` values in a single attribute called `info` in our `Movies` table, this attribute will be of DynamoDB map type. This illustrates how you can store JSON in a DynamoDB attribute.

The following is an example of movie data:

```
{
    "year" : 2013,
    "title" : "Turn It Down, Or Else!",
    "info" : {
        "directors" : [
            "Alice Smith",
            "Bob Jones"
        ],
        "release_date" : "2013-01-18T00:00:00Z",
        "rating" : 6.2,
        "genres" : [
            "Comedy",
            "Drama"
        ],
        "image_url" : "http://ia.media-imdb.com/images/N/O9ER
WAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@@._V1_SX400_.jpg",
        "plot" : "A rock band plays their music at high volumes, annoying the
neighbors.",
        "rank" : 11,
        "running_time_secs" : 5215,
        "actors" : [
            "David Matthewman",
            "Ann Thomas",
            "Jonathan G. Neff"
       ]
    }
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data file (`moviedata.json`) from the following location:
   - http://s3.amazonaws.com/getting-started-sample-data/moviedata.json

2. Copy the data file (`moviedata.json`) to a directory in your Java classpath.

Step 2.2: Load the Sample Data Into the Movies Table

After you have downloaded the data file, you can run the following program to load the data.

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
```

```
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import java.io.File;
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

public class MoviesLoadData {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        JsonParser parser = new JsonFactory()
            .createParser(new File("moviedata.json"));

        JsonNode rootNode = new ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();

        ObjectNode currentNode;
        while (iter.hasNext()) {
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();

            System.out.println("Adding movie: " + year + " " + title);

            table.putItem(new Item()
                .withPrimaryKey("year", year, "title", title)
                .withJSON("info", currentNode.path("info").toString()));
        }
        parser.close();
    }
}
```

This program uses the open source Jackson library to process JSON. Jackson is included in the AWS SDK for Java. You do not need to install it separately.

2. Compile and run the program.

Next Step

Step 3: Put, Update, and Delete an Item (p. 33)

Step 3: Put, Update, and Delete an Item

In this step, you will perform various operations on the items in the `Movies` table. You will run a Java program that demonstrates several operations on items in the table.

Topics

- Step 3.1: Add a New Item (PutItem) (p. 33)
- Step 3.2: Add an Item, Specify Condition (PutItem) (p. 34)
- Step 3.3: Update the Item (UpdateItem) (p. 35)
- Step 3.4: Increment an Atomic Counter (UpdateItem) (p. 37)
- Step 3.5: Update Item with a Condition (UpdateItem) (p. 38)
- Step 3.6: Delete an Item (DeleteItem) (p. 39)

Step 3.1: Add a New Item (PutItem)

In this step you add a new item to the `Movies` table.

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;

public class MoviesItemOps01 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        table.putItem(new Item()
            .withPrimaryKey("year", year, "title", title)
            .withJSON("info", "{\"plot\" : \"Something happens.\"}"));
        System.out.println("PutItem succeeded: " +
            table.getItem("year", year, "title", title).toJSONPretty());
    }
}
```

Note
The primary key is required. This code adds an item that will have primary key (`year`, `title`), and `info` attributes. The `info` attribute stores sample JSON that provides more information about the movie.

2. Compile and run the program.

Step 3.2: Add an Item, Specify Condition (PutItem)

By default, the `PutItem` operation replaces any existing item with a new item. The DynamoDB `PutItem` operation provides `ConditionExpression` to help you avoid overwriting an existing item. At present, the following item exists in the table. For details about `PutItem`, see PutItem in the Amazon DynamoDB API Reference.

```
{
  year: 2015,
  title: "The Big New Movie",
  info: {
          plot: "Something happens." }
 }
```

The code attempts to overwrite the item as follows:

```
{
   year: 2015,
   title: "The Big New Movie",
   info: {
        plot: "Nothing happens at all.",
        rating: 0
   }
}
```

For illustration, the `PutItem` operation in the following Java example specifies a condition that will fail because there is already an item with same primary key in the table.

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import java.util.HashMap;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.PrimaryKey;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.PutItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;

public class MoviesItemOps02 {
```

```
    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        final Map<String, Object> infoMap = new HashMap<String, Object>();
        infoMap.put("plot",  "Nothing happens at all.");
        infoMap.put("rating",  0.0);
        Item item = new Item()
            .withPrimaryKey(new PrimaryKey("year", year, "title", title))
            .withMap("info", infoMap);

        // Attempt a conditional write.  We expect this to fail.

        PutItemSpec putItemSpec = new PutItemSpec()
            .withItem(item)
           .withConditionExpression("attribute_not_exists(#yr) and attrib
ute_not_exists(title)")
            .withNameMap(new NameMap()
                .with("#yr", "year"));

        System.out.println("Attempting a conditional write...");

        table.putItem(putItemSpec);
        System.out.println("PutItem succeeded: " + table.getItem("year",
year, "title", title).toJSONPretty());
    }
}
```

2. Compile and run the program. It should fail with the following message: `The conditional request failed`
3. If you remove `ConditionExpression` in the preceding code, `PutItemSpec` will look as shown in the following code example and the code will now overwrite the item. Modify the program so that the `PutItemSpec` looks like this:

```
PutItemSpec putItemSpec = new PutItemSpec()
    .withItem(item);
```

4. Compile and run the program. The `PutItem` operation should now succeed.

Step 3.3: Update the Item (UpdateItem)

You can use the DynamoDB `UpdateItem` operation to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes in an `UpdateItem` operation. For details about `UpdateItem`, see UpdateItem in the Amazon DynamoDB API Reference.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The item will change from:

```
{
   year: 2015,
   title: "The Big New Movie",
   info: {
        plot: "Nothing happens at all.",
        rating: 0
   }
}
```

To the following:

```
{
   year: 2015,
   title: "The Big New Movie",
   info: {
           plot: "Everything happens all at once.",
           rating: 5.5,
           actors: ["Larry", "Moe", "Curly"]
   }
}
```

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.
package com.amazonaws.codesamples.gsg;

import java.util.Arrays;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class MoviesItemOps03 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";
```

```
        UpdateItemSpec updateItemSpec = new UpdateItemSpec()
            .withPrimaryKey("year", year, "title", title)
            .withUpdateExpression("set info.rating = :r, info.plot=:p, 
info.actors=:a")
            .withValueMap(new ValueMap()
                .withNumber(":r", 5.5)
                .withString(":p", "Everything happens all at once.")
                .withList(":a", Arrays.asList("Larry","Moe","Curly")));

        System.out.println("Updating the item...");

        table.updateItem(updateItemSpec);
        System.out.println("UpdateItem succeeded: " + table.getItem("year",
 year, "title", title).toJSONPretty());
    }
}
```

Note
Note the following about the `UpdateItemSpec` in the code:

- We use `UpdateExpression` to describe all updates you want to perform on the specified item.
- We specify the `ReturnValues` parameter to request DynamoDB to return only the updated attributes (`"UPDATED_NEW"`).

2. Compile and run the program.

Step 3.4: Increment an Atomic Counter (UpdateItem)

DynamoDB supports atomic counters, where you use the `UpdateItem` operation to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they were received.) For example, a music player application might want to maintain a counter each time song is played. In this case, the application would need to increment this counter regardless of its current value. For more information, see Atomic Counters in the Amazon DynamoDB Developer Guide. For details about `UpdateItem`, see UpdateItem in the Amazon DynamoDB API Reference.

The following program shows how to increment the `rating` for a movie. Each time you run it, the program increments this attribute by one.

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
```

```
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class MoviesItemOps04 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        UpdateItemSpec updateItemSpec = new UpdateItemSpec()
            .withPrimaryKey("year", year, "title", title)
            .withUpdateExpression("set info.rating = info.rating + :val")
            .withValueMap(new ValueMap()
                .withNumber(":val", 1));

        System.out.println("Incrementing an atomic counter...");

        table.updateItem(updateItemSpec);
        System.out.println("UpdateItem succeeded: " + table.getItem("year",
 year, "title", title).toJSONPretty());
    }
}
```

2. Compile and run the program.

Step 3.5: Update Item with a Condition (UpdateItem)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed. In this case, the item is only updated if there are more than three actors. For details about `UpdateItem`, see UpdateItem in the Amazon DynamoDB API Reference.

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.PrimaryKey;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
```

```
public class MoviesItemOps05 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        // Conditional update (will fail)

        UpdateItemSpec updateItemSpec = new UpdateItemSpec()
            .withPrimaryKey(new PrimaryKey("year", year, "title", title))
            .withUpdateExpression("remove info.actors[0]")
            .withConditionExpression("size(info.actors) > :num")
            .withValueMap(new ValueMap().withNumber(":num", 3));

        System.out.println("Attempting a conditional update...");

        table.updateItem(updateItemSpec);
       System.out.println("UpdateItem succeeded: " + table.getItem("year",
 year, "title", title).toJSONPretty());
    }
}
```

2. Compile and run the program. It should fail with the following message: `The conditional request failed`
3. Modify the program so that the `ConditionExpression` looks like this:

```
            .withConditionExpression("size(info.actors) >= :num")
```

   The condition is now *greater than or equal to 3* instead of *greater than 3*.
4. Compile and run the program. The `UpdateItem` operation should now succeed.

Step 3.6: Delete an Item (DeleteItem)

You can use the `DeleteItem` operation to delete one item by specifying its primary key. You can optionally specify a condition in the `DeleteItem` request to prevent item deletion based on a specified condition. In this following example, you will attempt to delete a specific movie item if its rating is 5.0 or less. For details about `DeleteItem`, see DeleteItem in the Amazon DynamoDB API Reference.

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.
```

```
package com.amazonaws.codesamples.gsg;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.PrimaryKey;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.DeleteItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class MoviesItemOps06 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        // Conditional delete (will fail)

        DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
            .withPrimaryKey(new PrimaryKey("year", 2015, "title", "The Big
 New Movie"))
            .withConditionExpression("info.rating <= :val")
            .withValueMap(new ValueMap()
                   .withNumber(":val", 5.0));

        System.out.println("Attempting a conditional delete...");

        table.deleteItem(deleteItemSpec);
        System.out.println("DeleteItem succeeded");
    }
}
```

2. Compile and run the program. It should fail with the following message because the rating for the particular move is greater than 5.0: `The conditional request failed`.
3. Now, modify the program to remove the condition in `deleteItemSpec`.

```
        DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
            .withPrimaryKey(new PrimaryKey("year", 2015, "title", "The Big
 New Movie"));
```

4. Compile and run the program. The delete will succeed because you removed the condition on `deleteItemSpec`.

Next Step

Step 4: Query and Scan the Data (p. 41)

Step 4: Query and Scan the Data

DynamoDB provides `Query` operation that you can use to query a table. The `Query` operation requires the hash attribute of the primary key; the range attribute is optional. For details about `Query`, see Query in the Amazon DynamoDB API Reference. In our example, the primary key for the `Movies` table is composed of the following hash and range attributes:

- `year`—the hash attribute of number type.
- `title`—the range attribute of string type.

To find all movies released during a year, you need to specify only the `year` hash attribute. You can add the `title` range attribute to retrieve a subset of movies based on some condition (on the range attribute). For example, to find movies released in 2014 that have a title starting with the letter "A".

DynamoDB also supports the `Scan` operation to read all the table data. For details about `Scan`, see Scan in the Amazon DynamoDB API Reference.

Topics

- Step 4.1: Query (p. 41)
- Step 4.2: Scan (p. 43)

Step 4.1: Query

The Java code included in this step performs the following queries:

- Retrieve all movies release in `year` 1985.
- Retrieve all movies released in `year` 1992, with `title` beginning with the letter "A" through the letter "L".

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import java.util.HashMap;
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;

public class MoviesQuery {

    public static void main(String[] args) throws Exception {
```

```
        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        HashMap<String, String> nameMap = new HashMap<String, String>();
        nameMap.put("#yr", "year");

        HashMap<String, Object> valueMap = new HashMap<String, Object>();
        valueMap.put(":yyyy", 1985);

        QuerySpec querySpec = new QuerySpec()
            .withKeyConditionExpression("#yr = :yyyy")
            .withNameMap(new NameMap().with("#yr", "year"))
            .withValueMap(valueMap);

        ItemCollection<QueryOutcome> items = table.query(querySpec);

        Iterator<Item> iterator = items.iterator();
        Item item = null;

        System.out.println("Movies from 1985");
        while (iterator.hasNext()) {
            item = iterator.next();
            System.out.println(item.getNumber("year") + ": " + item.get
String("title"));
        }

        valueMap.put(":yyyy", 1992);
        valueMap.put(":letter1", "A");
        valueMap.put(":letter2", "L");

        querySpec
            .withProjectionExpression("#yr, title, info.genres, info.act
ors[0]")
            .withKeyConditionExpression("#yr = :yyyy and title between
:letter1 and :letter2")
            .withNameMap(nameMap)
            .withValueMap(valueMap);

        items = table.query(querySpec);
        iterator = items.iterator();

        System.out.println("Movies from 1992 - titles A-L, with genres and
 lead actor");
        while (iterator.hasNext()) {
            item = iterator.next();
            System.out.println(item.toString());
        }
    }
}
```

Note

- `nameMap` provides name substitution. We use this because `year` is a reserved word in DynamoDB, you cannot use it directly in any expression, including `KeyConditionExpression`. We use the expression attribute name `#yr` to address this.

- `valueMap` provides value substitution. We use this because you cannot use literals in any expression, including `KeyConditionExpression`. We use the expression attribute value `:yyyy` to address this.

First, you create the `querySpec` object, which describes the query parameters, and then you pass the object to the `query` method.

2. Compile and run the program.

Note
The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information about secondary indexes, see Secondary Indexes in the Amazon DynamoDB Developer Guide.

Step 4.2: Scan

You will now perform a `Scan` operation on the table. A `Scan` operation reads every item in the entire table, and returns all of the data in the table, by default. You can optionally specify a filter expression for the results, so that only the items matching your criteria are returned. However, note that the filter is only applied after the entire table has been scanned.

The following Java program scans the entire `Movies` table, which contains approximately 5,000 items. The scan specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all of the others.

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.ScanSpec;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class MoviesScan {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");
```

```
        ScanSpec scanSpec = new ScanSpec()
            .withProjectionExpression("#yr, title, info.rating")
            .withFilterExpression("#yr between :start_yr and :end_yr")
            .withNameMap(new NameMap().with("#yr",  "year"))
            .withValueMap(new ValueMap().withNumber(":start_yr",
1950).withNumber(":end_yr", 1959));

        ItemCollection<ScanOutcome> items = table.scan(scanSpec);

        Iterator<Item> iter = items.iterator();
        while (iter.hasNext()) {
            Item item = iter.next();
            System.out.println(item.toString());
        }
    }
}
```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
- `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.

2. Compile and run the program.

Note
You can use the `Scan` operation with any secondary indexes that you have created on the table. For more information about secondary indexes, see Secondary Indexes in the Amazon DynamoDB Developer Guide.

Next Step

Step 5: (Optional) Delete the Table (p. 44)

Step 5: (Optional) Delete the Table

In this step, you will delete the `Movies` table.

This is an optional step. If you want, you can keep the `Movies` table and write your own programs to work with the data.

1. Copy the following program into your Java development environment.

```
// Copyright 2012-2015 Amazon.com, Inc. or its affiliates. All Rights Re
served.
// Licensed under the Apache License, Version 2.0.

package com.amazonaws.codesamples.gsg;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
```

```
public class MoviesDeleteTable {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        client.setEndpoint("http://localhost:8000");
        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");
        table.delete();

        System.out.println("Table status: " + table.getDescription().get
TableStatus());
    }
}
```

2. Compile and run the program.

Next Step

Summary (p. 45)

Summary

Topics

- Using the Amazon DynamoDB Service (p. 45)
- Next Steps (p. 53)

In this tutorial, you created the `Movies` table in DynamoDB Local and performed basic operations. DynamoDB Local is useful during application development and testing. However, when you are ready to run your application in a production environment, you need to modify your code so that it uses the Amazon DynamoDB service.

Using the Amazon DynamoDB Service

You will need to change the endpoint in your application in order to use the Amazon DynamoDB service as follows. To do this, first remove the following line:

```
        client.setEndpoint("http://localhost:8000");
```

Next, add a new line that specifies the AWS region you want to access:

```
        client.setRegion(Regions.REGION);
```

For example, if you want to access the `us-west-2 region`, you would do this:

```
        client.setRegion(Regions.US_WEST_2);
```

Now, instead of using DynamoDB Local, the program will use the DynamoDB service endpoint in US West (Oregon). For a list of available AWS regions, see Regions and Endpoints in the Amazon Web Services General Reference.

For more information about setting regions and endpoints in your code, see AWS Region Selection in the AWS SDK for Java Developer Guide.

DynamoDB Local is for development and testing purposes only. By comparison, DynamoDB is a managed service with scalability, availability, and durability features that make it ideal for production usage. The following table contains some other key differences between DynamoDB Local and the DynamoDB service:

Operation	DynamoDB Local	Amazon DynamoDB
Creating a table	The table is created immediately.	Table creation takes some time, depending on its provisioned throughput settings. DynamoDB allocates sufficient resources to meet your specific read and write capacity requirements.

Operation	DynamoDB Local	Amazon DynamoDB
Provisioned throughput	DynamoDB Local ignores provisioned throughput settings.	

Operation	DynamoDB Local	[illegible]
		[illegible]

Operation	DynamoDB Local	Amazon DynamoDB
		Working with Tables in the Amazon DynamoDB Developer Guide.

Operation	DynamoDB Local	Amazon DynamoDB
Reading and writing data	Reads and writes are performed as fast as possible, without any network overhead.	

Operation	DynamoDB Local	Amazon DynamoDB
		Read and write activity is regulated by the provisioned throughput settings on the table. To increase the maximum throughput, you must increase the throughput settings on the table. Network latency will also affect throughput to an

Operation	DynamoDB Local	Amazon DynamoDB
		extent.
Deleting a table	The table is deleted immediately.	Table deletion takes some time, as DynamoDB releases the resources that had been used by the table.

Next Steps

For more information about Amazon DynamoDB, see the Amazon DynamoDB Developer Guide. We recommend the following topics:

Data Model

Provisioned Throughput

Improving Data Access with Secondary Indexes

The Amazon DynamoDB Developer Guide also includes the following topics about working with tables, items, and queries:

Working with Tables

Working with Items

Query and Scan Operations in DynamoDB

Best Practices