JAVAWORLD

OPEN SOURCE JAVA TUTORIALS
By Steven Haines, Contributor, JavaWorld
JAN 16, 2018 11:09 AM PST

About
A working developer's guide to open source tools and frameworks for Java application development.

HOW-TO

Serverless computing with AWS Lambda, Part 2

Integrate AWS Lambda with DynamoDB, then call Lambda functions from a Java client

The first half of this article presented an overview of serverless computing with AWS Lambda, including building, deploying, and testing AWS Lambda functions in an example Java application. In Part 2, you'll learn how to integrate Lambda functions with an external database, in this case DynamoDB. We'll then use the AWS SDK to invoke Lambda functions from our example Java application.

AWS Lambda and DynamoDB

DynamoDB is a NoSQL document store that is hosted by Amazon Web Services (AWS). DynamoDB defines data abstractions as tables, which accept common database operations such as insert, retrieve, query, update, and delete. As with many other NoSQL databases, DynamoDB's schema isn't fixed, so some items in the same table can have fields that others do not.

One of DynamoDB's best features is its tiered pricing model. Unlike the AWS Relational Database Service (RDS), in which AWS manages your database using EC2 instances that you pay for, DynamoDB is pay-as-you-go. You pay for the storage you use and the throughput of your queries, but you don't directly pay for any underlying virtual machines. Additionally, AWS gives you a free tier supporting up to 25 GB of space, with enough throughput to execute up to 200 million requests per month.

In Serverless computing with AWS Lambda, Part 1, we developed a simple, serverless Java application using Lambda functions. You can download the source code for the GetWidgetHandler application anytime. If you haven't already read Part 1, I suggest familiarizing yourself with the application code and examples from that article before proceeding.

Our first step is to setup the DynamoDB database in our AWS console. After that we'll update the `get-widget` function from Part 1 to retrieve a widget from a DynamoDB table.

Setup the DynamoDB database in AWS

We'll start by creating the DynamoDB table. From the AWS console, click on **Services** and choose DynamoDB from the database section, as shown in Figure 1.

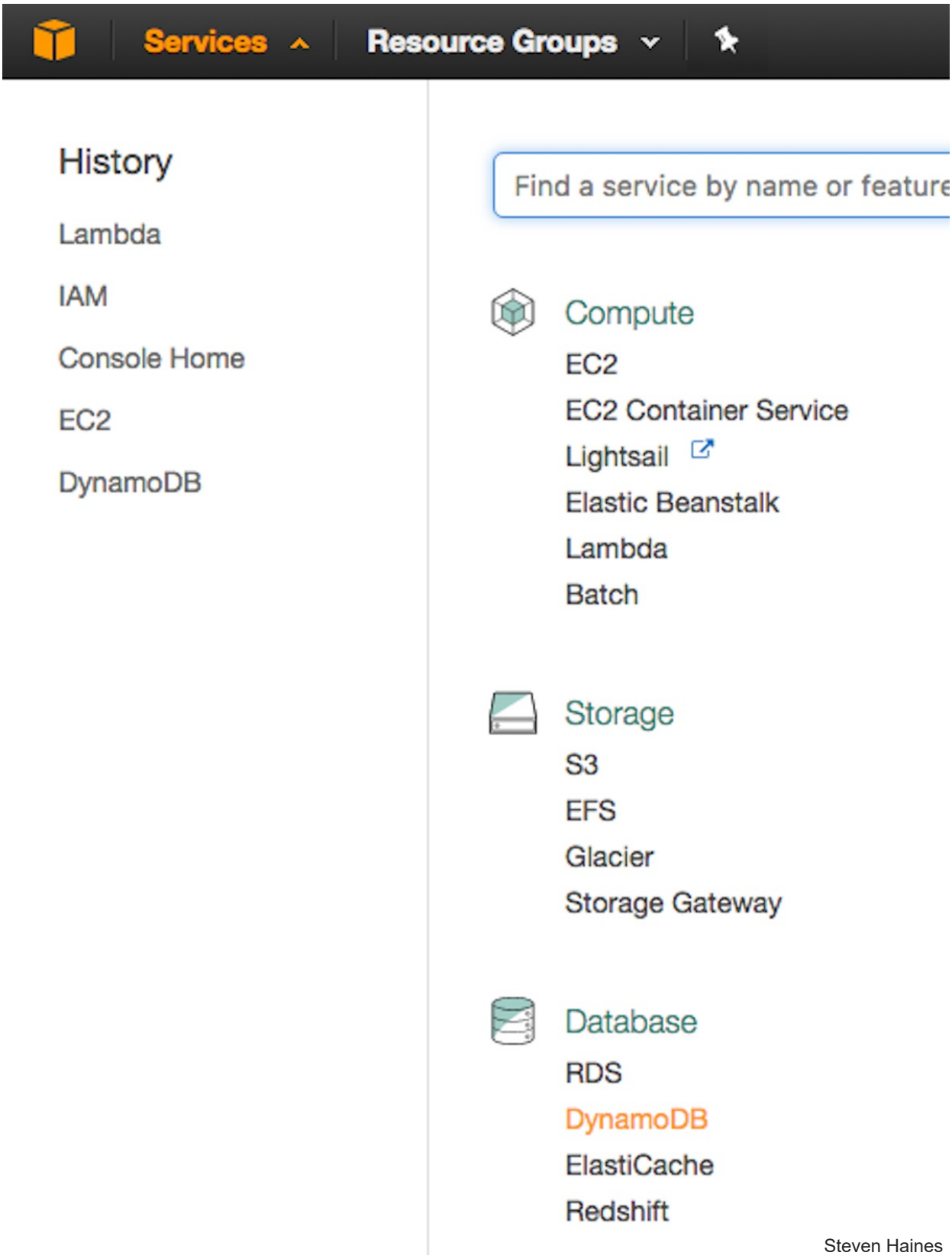

Figure 1. Launch the DynamoDB dashboard

Once launched, you'll see the DynamoDB dashboard. Click the **Create table** button to start creating your table, shown in Figure 2.

[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!]

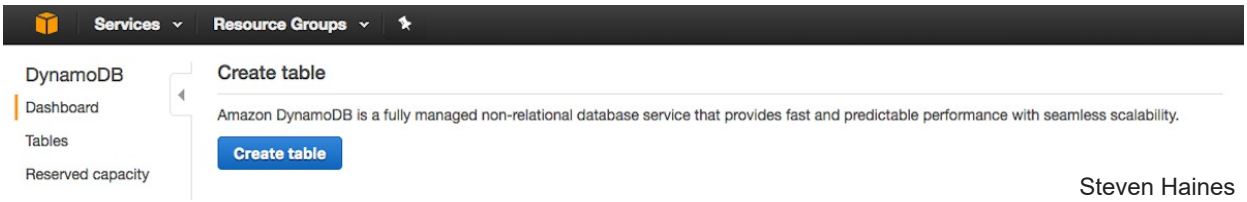

Figure 2. Create a DynamoDB table

Now you'll see the page shown in Figure 3.

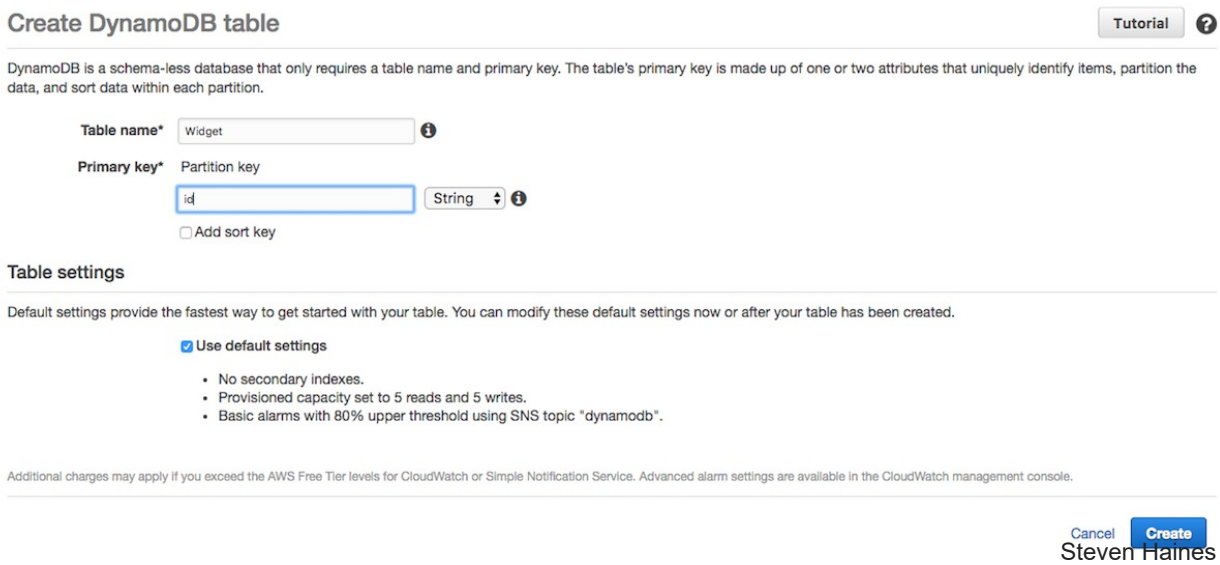

Figure 3. Create a DynamoDB table

Give your table a name (in this case "Widget") and set the primary key to `id`, leaving it as a `String`. Pressing **Create** when you are finished will direct you to the DynamoDB tables page. If you need to navigate to this page in the future, select **Services-->DynamoDB**, and click on **Tables**.

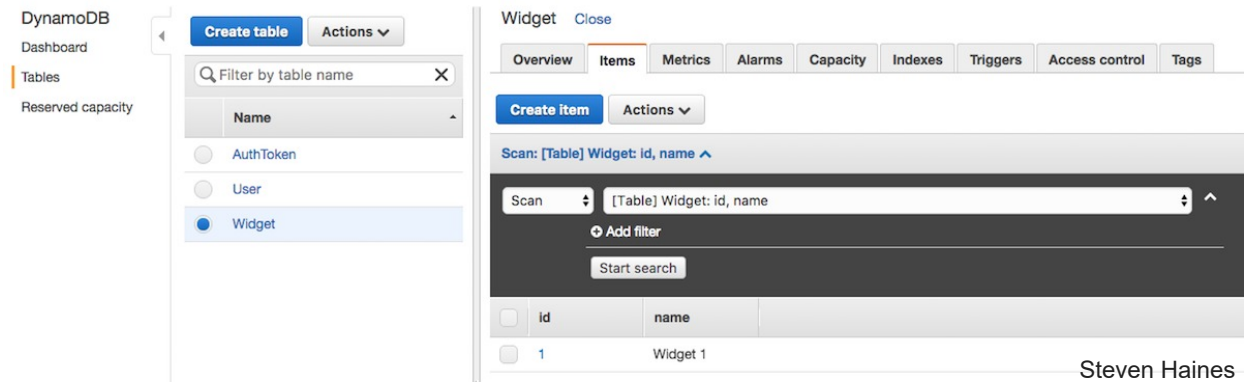

Figure 4. DynamoDB tables page

We'll manually create an entry in the new Widget table, so click the **Create item** button shown in Figure 5.

Figure 5. Create item page

DynamoDB will pre-populate the Create Item page with the `id` field. Enter an ID that is easy to remember, such as "1". Next, press the plus (+) next to the new ID, adding another field called `name`. Enter a value for the `name` field, such as "Widget 1". Press **Save** when you are finished.

Update the GetWidgetHandler class

With data in our database, the next thing we need to do is update the `GetWidgetHandler` class from Part 1. We'll start by adding the DynamoDB dependency to our original POM file. The updated `pom.xml` file is shown in Listing 1.

Listing 1. pom.xml (updated with DynamoDB dependency)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.javaworld.geekcap</groupId>
    <artifactId>aws-lambda-java</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>aws-lambda-java</name>
    <url>http://maven.apache.org</url>

    <properties>
        <java.version>1.8</java.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>


    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.1.0</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-java-sdk-dynamodb</artifactId>
            <version>1.11.135</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.3</version>
```

```
            <configuration>
                <createDependencyReducedPom>false</createDependencyReducedPom>
            </configuration>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
</project>
```

Listing 1 adds the `aws-java-sdk-dynamodb` dependency to the POM file from Part 1. Listing 2 shows the updated `GetWidgetHandler` class.

Listing 2. GetWidgetHandler.java (updated to load data from DynamoDB)

```
package com.javaworld.awslambda.widget.handlers;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.javaworld.awslambda.widget.model.Widget;
import com.javaworld.awslambda.widget.model.WidgetRequest;

public class GetWidgetHandler implements RequestHandler<WidgetRequest, Widget> {
    @Override
    public Widget handleRequest(WidgetRequest widgetRequest, Context context) {
        //return new Widget(widgetRequest.getId(), "My Widget " + widgetRequest.getId());

        // Create a connection to DynamoDB
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.defaultClient();
        DynamoDB dynamoDB = new DynamoDB(client);

        // Get a reference to the Widget table
        Table table = dynamoDB.getTable("Widget");

        // Get our item by ID
        Item item = table.getItem("id", widgetRequest.getId());
        if(item != null) {
            System.out.println(item.toJSONPretty());

            // Return a new Widget object
            return new Widget(widgetRequest.getId(), item.getString("name"));
        }
        else {
            return new Widget();
        }
    }
}
```

The main interface to DynamoDB is the `DynamoDB` object. In order to create a `DynamoDB` instance, we need a DynamoDB client. Because our Lambda function will run in AWS, we do not need to provide credentials, so we can use the default client. Note that we'll only be able to query the database without credentials because the `get-widget-role` from Part 1 has the `dynamodb:GetItem` permission.

From the `DynamoDB` instance, we can call `getTable("Widget")` to retrieve a `Table` instance. Then we can call `getItem()` on the `Table` instance, passing it the primary key of the item we want to retrieve. If there is an item with the specified primary key then it will return a valid response; otherwise it will return `null`. The `Item` class provides access to the response parameters, so we finish up the implementation by creating a new `Widget` object with its name loaded from DynamoDB.

Get the code

Get the code for the updated GetWidgetHandler application. *Created by Steven Haines for JavaWorld.*

Querying DynamoDB with DynamoDBMapper

There are several APIs for querying DynamoDB, from a RESTful service call, to the low-level interface above, to a couple of higher level interfaces. One of the more popular interfaces is DynamoDBMapper. This interface provides a similar construct to what you might find when mapping objects to relational data in a tool like Hibernate. Let's briefly review how to retrieve a `Widget` from DynamoDB using the `DynamoDBMapper` API.

The first thing that we need to do is add a few annotations to the `Widget` class, which is shown in Listing 3.

Listing 3. Widget.java (updated with DynamoDBMapper annotations)

```
package com.javaworld.awslambda.widget.model;

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName="Widget")
public class Widget {
    private String id;
    private String name;

    public Widget() {
    }

    public Widget(String id) {
        this.id = id;
    }

    public Widget(String id, String name) {
        this.id = id;
        this.name = name;
    }

    @DynamoDBHashKey(attributeName="id")
    public String getId() {
        return id;
    }

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

    @DynamoDBAttribute(attributeName="name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

The `DynamoDBTable` annotation specifies the name of the DynamoDB table to which the `Widget` maps. The `DynamoDBHashKey` annotation identifies the primary key of the `Widget` table. And the `DynamoDBAttribute` annotation identifies other class attributes that map to database attributes in DynamoDB. If you had other attributes that you wanted to ignore, you could add the `@DynamoDBIgnore` annotation.

With the `Widget` class annotated, we can now update the `GetWidgetHandler` class to use the `DynamoDBMapper`, which is shown in Listing 4.

Listing 4. GetWidgetHandler.java (updated with DynamoDBMapper)

```
package com.javaworld.awslambda.widget.handlers;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.javaworld.awslambda.widget.model.Widget;
import com.javaworld.awslambda.widget.model.WidgetRequest;

public class GetWidgetHandler implements RequestHandler<WidgetRequest, Widget> {
    @Override
    public Widget handleRequest(WidgetRequest widgetRequest, Context context) {
        // Create a connection to DynamoDB
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.defaultClient();

        // Build a mapper
        DynamoDBMapper mapper = new DynamoDBMapper(client);

        // Load the widget by ID
        Widget widget = mapper.load(Widget.class, widgetRequest.getId());
        if(widget == null) {
            // We did not find a widget with this ID, so return an empty Widget
            context.getLogger().log("No Widget found with ID: " + widgetRequest.getId() + "\n");
            return new Widget();
        }

        // Return the widget
        return widget;
    }
}
```

In the former (Part 1) version of the `GetWidgetHandler` we created an `AmazonDynamoDB` instance, using a `AmazonDynamoDBClientBuilder.defaultClient()` call. Now we'll use that client to initialize a `DynamoDBMapper` instance instead.

The DynamoDBMapper class provides access to execute queries, load objects by ID, save objects, delete objects, and so forth. In this case, we pass `DynamoDBMapper` the widget's class (`Widget.class`) and its primary key. If DynamoDB has a `Widget` with the specified

primary key it will return it; if not it will return null.

Rebuild and then re-upload your new JAR file by opening your Lambda function dashboard, then click on the **Code** tab and press **Upload**. When you re-upload and subsequently call your function, AWS Lambda will create a new container for the new JAR file and push that to an EC2 instance. You should expect the first run to be slow.

If you happen to encounter an `OutOfMemoryError` when you re-test your function, select the **Configuration** tab and open the Advanced Settings section. Here you can increase your memory, as shown below.