## Exercise 2

# Exercise: Creating an Amazon DynamoDB Table using the AWS Software Development Kit (AWS SDK)

**Overview**

In this exercise, you will learn how to *develop* with Amazon DynamoDB by using the AWS Software Development Kit (AWS SDK). Following the scenario provided, you will create a DynamoDB table This exercise gives you hands-on experience with both Amazon DynamoDB and AWS Cloud9.

**Objectives**

After completing this exercise, you will be able to use the AWS SDKs to do the following:

* Create a DynamoDB table.

**Story**

One of your best friends (Mary) who you've known for years has come to you for help.

Mary has always been obsessed with fantasy dragon card games, and she has recently set up a gaming company and her latest project is a new dragon card game being developed internally by a local group of enthusiasts (who are now her employees).

They don't have a website for their game yet. They have been doing everything manually, even drawing the cards themselves.

The problem is none of them are very tech savvy, which is why Mary reached out to you.

There is currently no way to look up card details online.

Mary has asked you to help her create a simple webpage that can shows all the dragon cards when the page loads, and to be able show any card and all of it's information by doing a simple search on a dragon name.

You don't even ask what budget she has, as you already know the answer. Zero.

Beside you want the practice, as you are sitting your developing on AWS exam shortly.

**The plan**

You figure that a basic HTML page with a search box, and some CSS and a bit of AJAX to a database would probably work as a prototype. You decide to set that up on your own account inside your AWS free tier.

You recently took a serverless course, and feel obliged to make this whole serverless just out of principle.

As you are planning on sitting your AWS developer associate exam shortly. You decide to force yourself to do most of these activities using the AWS-SDK, and get comfortable with code.

You think the best place to start is to create a DynamoDB table with an open schema. This way you can decide on the schema once you know what the data looks like, and how you plan on querying the data.

You imagine it will look a bit like this, as dragon names are unique.

| **Primary Key (dragon\_name)** | **<attr>** |
| --- | --- |
| sparky | ... |
| tallie | ... |

**Prepare the exercise**

Before you can start this exercise, you need to import some files and install some modules in the AWS Cloud9 environment that you will create.

1. From the AWS Management Console, go to the **Services** menu and choose **Cloud9**.
2. Choose **Create environment**.
3. Under **Name**, input dynamolab.
4. Choose **Next step**, **Next step**, **Create environment**.
5. Choose **Open IDE** to open the AWS Cloud9 environment.
6. To seed your Cloud9 filesystem. Go to the Cloud9 **bash terminal** (at the bottom of the page) and run the following wget command: *You should be in /home/ec2-user/environment*

wget https://s3.amazonaws.com/awsu-hosting/edx\_dynamo/c9/dynamo-create/lab1.zip -P /home/ec2-user/environment

You should also see that a root folder called **dynamolab** with a lab1.zip file has been downloaded and added to your AWS Cloud9 filesystem (on the top left). There is also a README.md file in there by default, which you will remove in a bit.

1. To unzip the lab1.zip file, run the following command:

unzip lab1.zip

1. To keep things clean, run the following commands to remove the zip and README files. Install any dependencies in the root and node folder. Then navigate to the lab1 folder path (*/home/ec2-user/environment/lab1*) in the terminal by using these commands.

rm lab1.zip

rm README.md

cd lab1

npm install aws-sdk

echo "done"

1. Once you see "done", select the black arrow next to the lab1 folder (far left top) to expand it. Notice that there is a solution folder. **Try not to peek at the solution unless you really get stuck. Always TRY to code first.**
2. You should see that some packages and modules have been installed. Ignore any warnings in the terminal.

⚠️

*As this course is self paced, often people will start the lab then come back to it later. in the interim period we may have made adjustments to the code.*

*Ensure that inside your lab1 folder that the name of the version markdown file is matching the version number at the top of this document.*

If they are out of sync, you will run into problems, to get them synced simply remove the old folder and run through the wget steps above one more time,

You are now ready to do the exercise tasks.

**Step 1: Create a DynamoDB table using the AWS SDK**

You want to use the AWS SDK to create a DynamoDB table. You figure the method name is probably something a bit like **create table**, but you check the AWS SDK documentation anyway.

1. From the table below, open the link to the method for creating DynamoDB tables in the AWS SDK documentation. Confirm the method name and establish what parameters you need to pass in.

| **Language** | **AWS SDK Documentation deep link** |
| --- | --- |
| Node.js (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#createTable-property> |

**Time to write some code that creates a new DynamoDB table.**

In the AWS Cloud9 environment, do the following:

1. Open (double-click) the create\_table.js file inside the lab1 folder (not the one inside the solution folder).
2. Using the SDK documentation to help you, simply replace the sections of the code in that file so that the code will create a table called **dragons** in **us-east-1**.

*Tip: You think initially you will only need to search for a dragon by its name, so you figure that the best choice for the primary key will be dragon\_name with no sort key. You are not sure how often this table will be used so you think it best to use* ***dynamic capacity*** *rather than guessing the necessary provisioned capacity.*

1. Save the create\_table.js file.
2. **LAB TIP:** Before you run this script, you should double check the solution code first. This way when you run the file it will just work. It is very easy to end up with some parts of a script working and some not, due to a typo or a small mistake, and end up with partial or inaccurate resources being created. It is better to check your code meticulously against the solution code before actually executing your code, rather than trying to undo partial changes after the script has run.
3. In the Cloud9 terminal, type this run command.

node create\_table.js

**Confirm that your code worked.**

In the terminal you should see something a bit like this:

null { TableDescription:

  { AttributeDefinitions: [ [Object] ],

    TableName: 'dragons',

    KeySchema: [ [Object] ],

    TableStatus: 'CREATING',

    CreationDateTime: 2019-06-03T13:38:16.594Z,

    ProvisionedThroughput:

    { NumberOfDecreasesToday: 0,

       ReadCapacityUnits: 0,

       WriteCapacityUnits: 0 },

    TableSizeBytes: 0,

    ItemCount: 0,

    TableArn: 'arn:aws:dynamodb:us-east-1:000000000000:table/dragons',

    TableId: '4b78ab3a-288b-46a4-b3df-74395804dfbc',

    BillingModeSummary: { BillingMode: 'PAY\_PER\_REQUEST' } } }

We can also verify that the table was created in the DynamoDB console.

1. Choose **AWS Cloud9** in the upper left.
2. Then choose **Go To Your Dashboard**.
3. Choose **Services** and search for **dynamo** then choose **DynamoDB** to pivot to the console.
4. Choose **Tables**.

If your code worked, you will see that your table has been (or is being) created in the US East (N. Virginia) Region.

Select the name of your table. In the **Overview** tab, you should see something similar to this:

**IF YOU GET STUCK, OR YOUR CODE DOES NOT WORK AS EXPECTED, LOOK AGAIN AT THE SOLUTION CODE, OR REACH OUT TO US IN THE FORUMS.**

**Congratulations!** You have completed exercise 1. You have created a DynamoDB table in the N.Virginia region.

Now you are ready to add Dragons!

**Lab Tip**: We recommend cleaning up after yourself by closing down any tabs and collapse and folders that you are no longer working on in the Cloud environment.

## Exercise 2

[version\_1.0.3]

# Exercise: Adding items to Amazon DynamoDB using the AWS Software Development Kit (AWS SDK)

## Overview

In this exercise, you will learn how to develop with Amazon DynamoDB by using the AWS Software Development Kit (AWS SDK). Following the scenario provided, you will add items into an existing DynamoDB table and use the AWS SDK. This exercise gives you hands-on experience with Node.Js, Amazon DynamoDB, and AWS Cloud9.

## Objectives

After completing this exercise, you will be able to use the AWS SDKs to do the following:

* Upload items to the DynamoDB table.
* Query your DynamoDB table using code (i.e a full table scan).
* Create a role for a Lambda function using AWS Identity and Access Management (IAM).
* Create an AWS Lambda function that talks to DynamoDB, using the Lambda console.
* Create an CORS enabled Amazon API gateway that points to a Lambda function
* Upload an item to Amazon S3 via the AWS-SDK

## Story continued

Now you have your database ready, it's time to seed it with some data.

You asked Mary for some card data and she has promised to email you a JSON document with all the card data and some more dragon images.

However she keeps delaying. So you think it's a good idea to add a few items to help you create a basic proof of concept. The API will return all the data in the database to the website. You only have 2 images from her so far, so your database needs only to have 2 items

Your next step is to add a couple of dragon items to the database table that you just created.

You think dragon info would look a bit like this (you are clearly guessing). So you add this as a starting point while you wait for Mary to email you the real data.

###### Dragon table

| **Primary Key (dragon\_name)** | **drgaon\_type** | **description** | **attack** | **defense** |
| --- | --- | --- | --- | --- |
| sparky | green | breaths acid | 10 | 7 |
| tallie | red | breaths fire | 7 | 10 |

You already have a basic website that you put together over the weekend, however you haven't built the back end API functionality for it yet. We can do that now.

We will be taking this step by step.

1. Upload your basic website to S3 and configure it for website hosting
2. Upload some dragon data using the AWS-SDK.
3. Build out the backend functionality.

## Prepare the exercise

Before you can start this exercise, you need to import some files and install some modules in the AWS Cloud9 environment that you have created.

1. If you are not already in your Cloud 9 environment (from the last lab). Go to the AWS Management Console. Choose the **Services** menu and choose **Cloud9**, and choose **Open IDE** to open the AWS Cloud9 environment.
2. In the Cloud9 system go to the Cloud 9 **bash terminal** and type

cd /home/ec2-user/environment

1. If you had any lab1 stuff open, collapse that folder now, and kill off any tabs you are not using from the last lab. Things can get crowded very quickly in Cloud 9.
2. You will need to seed your Cloud9 filesystem, go to the Cloud 9 **bash terminal** (at the bottom of the page) and run the following wget command:

wget https://s3.amazonaws.com/awsu-hosting/edx\_dynamo/c9/dynamo-update/lab2.zip -P /home/ec2-user/environment

You should also see that a root folder called **dynamolab** with a lab2.zip file has been downloaded and added to your AWS Cloud9 filesystem (on the top left).

1. To unzip the lab2.zip file, run the following command:

unzip lab2.zip

1. To keep things clean, run the following commands to remove the zip files:

rm lab2.zip

cd lab2

echo "done"

1. Once you see "done". Select the black arrow next to the lab2 folder (top left) to expand it. Notice inside this lab2 folder there is a solution folder. **Try not to peek at the solution unless you really get stuck. Always TRY to code first.**
2. You will notice a resources folder inside lab2 too (you can ignore this for now). We will use items in here later in the other exercises.
3. Run this command to bring in the SDK into the lab2 folder path.

npm install aws-sdk

You should see that some packages and modules have been installed. Ignore any warnings in the terminal. However, if you get an **error**, reach out in the forums before moving on.

⚠️

As this course is self paced, often people will start the lab then come back to it later. in the interim period we may have made adjustments to the code.

Ensure that inside your *lab2* folder that the name of the version markdown file is matching the version number at the top of this document.

If they are out of sync, you will run into problems, to get them synced simply remove the old folder and run through the wget steps above one more time,

You are now ready to do the exercise tasks.

## Step 1: Add Dragon data to your DynamoDB table using the SDK

We need to populate DynamoDB with some dummy dragon data so our website can eventually query it and display dragon data.

1. Open the SDK docs for node.js, and find the method for creating new items inside an existing DynamoDB table. Find out the correct method name and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#putItem-property> |

**Time to write some code that adds a couple of items to your DynamoDB table.**

1. In Cloud 9 open up the upload\_items file inside the lab2 folder by double clicking on it.
2. Have the SDK docs open (as above) to help you.
3. Replace the sections of the code in that file. So that the code uploads dragon data to the table you created in lab1. Your table should have been called **dragons** and should be in **us-east-1**.
4. Here is the dragon info you wish to upload. As you created a flexible schema you can drop in the following data structure.

| **Primary Key (dragon\_name)** | **dragon\_type** | **description** | **attack** | **defense** |
| --- | --- | --- | --- | --- |
| sparky | green | breaths acid | 10 | 7 |
| tallie | red | breaths fire | 7 | 10 |

1. Save the upload\_items.js file.
2. In the Cloud9 terminal. Type this run command, rather then just pressing the run button.

node upload\_items.js

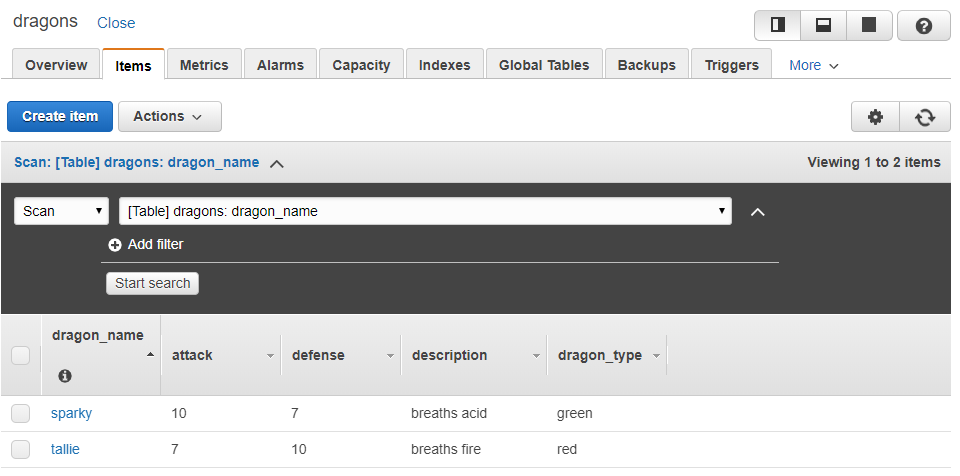
You should see something like this, showing two items have been added.

null { ConsumedCapacity: { TableName: 'dragons', CapacityUnits: 1 } }

null { ConsumedCapacity: { TableName: 'dragons', CapacityUnits: 1 } }

## Confirm that your code worked.

1. Head back to your DynamoDB console (probably still open in another browser tab from lab1). Refresh the table.
2. Click on your dragons table, and choose the **items** tab. Then press **scan**, you should see both dragon items (see image below).



**Congrats** you now have dummy Dragon data inside DynamoDB.

**IF YOU GET STUCK, SIMPLY COPY THE CODE SITTING IN THE SOLUTION FOLDER INTO YOUR FILE AND THEN RUN IT**.

## Step 2: Wiring up the data to a web front end.

You spent time over the weekend putting together a basic HTML front end to display dragon card data. You now need to upload it to S3 and configure it to be hosted as a website.

#### Step 2A): Upload the website to S3

We have prepared a script for you that will upload your website (currently sitting in that resources folder we talked about earlier).

You need to write no code, and you will not be asked to ever modify the website, however for this upload script to work you will need to provide a unique bucket name in your account. As well as an IP address from where you are developing this exercise.

1. Choose **Services** and search for **s3**.
2. Choose **Create bucket**.
3. For **Bucket name** make sure you type in something unique but easy to remember. **Example**: er-101-2019-05-16-dragon-website. Choose **Next**.
4. Choose **Next** again. Leave **Block all public access** checked and choose **Next**.
5. Choose **Create bucket**.
6. Once the bucket is created choose it from the S3 buckets list.
7. Choose the **Permissions** tab and select **Bucket Policy**.
8. Copy and paste the following policy below:
   1. Replace "Resource": "arn:aws:s3:::yourwebsite/\*", with your bucket name. Example: "Resource": "arn:aws:s3:::er-101/\*",
   2. Also browse to <https://www.whatismyip.com/> and replace "0.0.0.0" with your IPv4 address. This will only allow bucket access from your IP. So if you test this from another place, make sure the IP is updated.

{

   "Version": "2008-10-17",

   "Statement": [

      {

           "Effect": "Allow",

           "Principal": "\*",

           "Action": "s3:GetObject",

           "Resource": "arn:aws:s3:::yourwebsite/\*",

           "Condition": {

               "IpAddress": {

                   "aws:SourceIp": [

                       "0.0.0.0/32"

                  ]

              }

          }

      }

  ]

}

1. Choose **Save**.
2. Choose the **Properties** tab and choose **Static website hosting**.
3. Choose **Use this bucket to host a website**.
4. Type index.html in the Index document field.
5. Type error.html in the Error document field.
6. Choose **Save**.
7. Choose **Static website hosting** again and you will see the Endpoint similar to this: <http://2019-05-16-dynamolab-er-102.s3-website-us-east-1.amazonaws.com>. Click it to open it up in a new browser tab.

If you browse to the endpoint you will see **404 Not Found**. This is expected as you should have access to view the page but currently we don't have anything in our bucket.

Lab tip: Make a note of your bucket name somewhere, you will need it soon. We also suggest you leave this particular website tab open

**\*NOTE**: Please use the chrome browser for viewing this dragon website, as we are not supporting other browsers in the forums.\*

#### Step 2B): Upload items

1. Head back to your **Cloud9** tab. You will see a resources folder in the lab 2 folder.
2. Close any tabs and collapse any folders that you are not using in the Cloud 9 IDE.
3. Double click on /lab2/resources/upload\_website.js.
4. You will only need to modify line 42 where it asks for a bucket name. You do not need to modify any other parts of this file. However if you are interested you can see the code.
5. Replace the (the fill me in) on line 42 with **your** bucket name. Example: "2019-05-16-dynamolab-er-102"
6. Choose **File** and **Save**.
7. Then run the following:
8. pushd /home/ec2-user/environment/lab2/resources && node upload\_website.js && popd

The output should look like the following:

null { ETag: '"07df88017f8e994a0787eafe4a0db357"' }

null { ETag: '"b209006f7bff76e010dca9ebd9437fe5"' }

null { ETag: '"1ac51b3c4111aed24f4496ab0b6b8abc"' }

.....<many more items>

We uploaded multiple files used in future labs to save you going through this step for every lab.

Now your website is all wired up to hit an API and return all the dragon data in the database as a proof of concept. The challenge is you don't have an API endpoint "to use" yet.

Go have a look at your website again.

So when you load the website it will say **"No API to call".** Which is to be expected, as you don't have one yet

However you thought ahead, and have created a config.js file for your website. This can easily be updated with the API endpoint once you have one.

You will come back to this later.

Meanwhile you will be asked to complete the following steps:

* Use Cloud 9 and create a script that queries your DynamoDB table. Test it to ensure that it returns all the dragon data it has (a full table scan).
* Create a role in IAM that can be used by a Lambda function that you are about to create that allows Lambda to talk to DynamoDB.
* Copy and paste the working code into the Lambda console, and using that role, create a working Lambda function and test it.
* Create a CORS enabled API gateway that points to your Lambda function, and test it.
* Upload a new config.js file containing your new API endpoint.
* Visit your website, and see all the dragon data (ready to show Mary as a proof of concept).

## Step 3: Create a scan script

We will first write and test a script (using Cloud 9) that will query your dragon database.

You remember that to scan a DynamoDB table you would use a method like **scan**, but you double check the AWS SDK documentation to make sure.

From the table below, open the link to the method for scanning a DynamoDB table in the AWS SDK documentation. Confirm the method name and establish what parameters you need to pass in.

| **Language** | **AWS SDK Documentation deep link** |
| --- | --- |
| Node.js (6.17.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#scan-property> |

In the AWS Cloud9 environment, do the following:

1. Close any Cloud 9 tabs you are not using such as upload\_website.jsand collapse the resources folder.
2. Open (double-click) the scan\_dragons file.
3. Using the AWS SDK documentation to help, replace the sections of the code in that file so that the code searches your **dragons** database and returns all the dragon data it has (a scan). Also modify the code to return the dragon information in a way the browser likes to receive data (as an array of data items ).
4. Save the file.
5. LAB TIP: Before you run this script, you should double check the solution code first. This way when you run the file it will just work. It is very easy to end up with some parts of a script working and some not, due to a typo or a small mistake, and end up with partial or inaccurate resources being created. It is better to check your code meticulously against the solution code before actually executing your code, rather than trying to undo partial changes after the script has run.
6. In the AWS Cloud9 terminal, run this command. (you should already be in the /lab2 path)
7. node scan\_dragons.js test

## Confirm that your code worked.

When you run the script, you should see following information.

Local test for all dragons

null [ { dragon\_type: { S: 'green' },

defense: { N: '7' },

description: { S: 'breaths acid' },

attack: { N: '10' },

dragon\_name: { S: 'sparky' } },

{ dragon\_type: { S: 'red' },

defense: { N: '10' },

description: { S: 'breaths fire' },

attack: { N: '7' },

dragon\_name: { S: 'tallie' } } ]

Next we need to create a Lambda function out of this. But first we will finish a few subtasks.

#### Step 3A): Create a new IAM role for use with Lambda

By creating a role for the Lambda function you are about to create, you are essentially allowing your function code to communicate freely with DynamoDB, write logs to CloudWatch Logs, report to AWS Xray and freely touch S3 (which we will need in future labs).

1. From your Cloud9 dashboard choose **AWS Cloud9** in the upper left.
2. Then choose **Go to your dashboard**.
3. Go to **services** and choose **IAM**.
4. Choose **Roles** and choose **create role**.
5. Select **Lambda** and choose **Next: Permissions**.
6. In the **search** box type in **Dynamo** and select the **checkbox** next to AmazonDynamoDBFullAccess.
7. Don't press next yet.
8. Clear the search box and again in the **search** box type in **Lambda** and select the **checkbox** next to AWSLambdaBasicExecutionRole.
9. And again in the **search** box type in **AWSXrayWriteOnlyAccess** and select the **checkbox** next to AWSXrayWriteOnlyAccess.
10. And again in the **search** box type in **AmazonS3FullAccess** and select the **checkbox** next to AmazonS3FullAccess.
11. Choose **Next: Tags** and leave it as is. Select **Next:Review**.
12. Type the name call-dynamodb-role in the **Role name** box. Then choose **Create role**.

Lab note: We only asked you to add Xray and full Dynamo and full S3 access to save you doing these steps again later in future labs.

Real world tip: In production environments you should lock down these permissions further and apply the principal of least principle.

#### Step 3B): Create a Lambda function

You already have your scan script ready and tested. So all you need to do now is create a Lambda function out of that code.

The code in the scan script was set up to work both in a Cloud9 testing environment and in a Lambda environment. You do not need to alter the code.

You just need to create a Lambda function, passing in that role you just created, then paste in your code "as is" from your existing **scan** script.

Just follow these steps:

1. Choose **services** and search for **lambda**.
2. Choose **lambda** from the drop-down list.
3. Choose **Create function**.
4. Type in DragonSearch for the **Function name**.
5. Use **Node.js 10.x** for the **Runtime**.
6. Under **Permissions** open **Choose or create an execution role**.
7. Under **Execution role** choose **Use an existing role**.
8. In the **Existing role** drop-down choose the role we created above call-dynamodb-role.
9. Finally choose **Create function**.
10. In the other browser tab where you have cloud9 open. Simply copy the code from scan\_dragons.js and paste it into the Lambda editor replacing the contents of index.js.
11. Scroll down and under **Basic settings**. Change the timeout to 0 mins and 10 sec.
12. Choose **Save** at the top of the page.

#### Step 3C): Test our function

1. Next to the **Test** button select the drop-down arrow.
2. Choose **Configure test events**.
3. Leave the **Event template** as **Hello World**. In the **Event name** box type in DragonScan and paste in the following **blank object** code:
4. {
5. }
6. Choose **Create**.
7. Choose **Test**.

You should see the following in the **Execution result** go green, and when you expand the details sections you should see something like this:

[

{

"dragon\_type": {

"S": "green"

},

"defense": {

"N": "7"

},

"description": {

"S": "breaths acid"

},

"attack": {

"N": "10"

},

"dragon\_name": {

"S": "sparky"

}

},

{

"dragon\_type": {

"S": "red"

},

"defense": {

"N": "10"

},

"description": {

"S": "breaths fire"

},

"attack": {

"N": "7"

},

"dragon\_name": {

"S": "tallie"

}

}

]

**Congrats!** Your Lambda function can scan DynamoDB and return data. Now we need to put the API endpoint in front of it, so you can use it in your website.

## Step 4: Create a simple API and connect it to your website

To create and test an API from the Amazon API Gateway console, do the following:

1. Choose **services** and search for **API Gateway**.
2. Choose **Get Started** and choose **OK** to remove the **Create Example API** pop-up.
   1. If you already have an API Gateway. Choose **Create API**.
3. Under **Choose the protocol**, select **REST**.
4. Under **Create new API**, select **New API**.
5. For settings, enter the following:

**API name**: DragonSearchAPI

**Description**: Add a brief description (optional)

**Endpoint Type**: Regional

1. Choose **Create API**.
2. Choose **Actions** and then **Create Method**.
3. From the drop-down list under the **/** under **Resources**, choose **POST** and select the check mark icon.
4. For **Integration type**, choose **Lambda Function**.
5. Make sure the Lambda Region is set to **us-east-1**.
6. Ensure that you **do not** select **Use Lambda Proxy integration**. That must remain **unchecked**.
7. Under Lambda Function start typing the word Dragon, so you can choose **DragonSearch** from the list.
8. Set **custom timeout** to 10 seconds 10000.
9. Click **Save**.
10. Click **OK** to bypass the pop-up telling us about giving permissions for API gateway to talk to Lambda. This is fine, and just what we want it to do.
11. Click **TEST** and you should see a page where you can type the "request body", which you leave **blank**.
12. Scroll to the bottom to press the **Test** button.

Under **Response Body** you should see something like the following:

[

{

"dragon\_type": {

"S": "green"

},

"defense": {

"N": "7"

},

"description": {

"S": "breaths acid"

},

"attack": {

"N": "10"

},

"dragon\_name": {

"S": "sparky"

}

},

{

"dragon\_type": {

"S": "red"

},

"defense": {

"N": "10"

},

"description": {

"S": "breaths fire"

},

"attack": {

"N": "7"

},

"dragon\_name": {

"S": "tallie"

}

}

]

## Important step

Before your website can talk to this back end API. We need to enable CORS before deploying it.

Here is why:

Your website is hosted in one domain (your S3 website URL) and your API is hosted in a different domain (the API Gateway endpoint URL). Your browser doesn't like this, and blocks the request.

You need to let your browser know that it is OK to call your API Gateway endpoint URL from your website. You do this by using what is known as a preflight request.

This entire process of allowing cross-domain access with preflight requests is called "Enabling CORS" (Cross-Origin Resource Sharing).

To enable CORS from the Amazon API Gateway console, do the following:

1. Click on the resource **/** above POST. You should see a green POST box appear on the right.
2. Choose **Actions** and select **Enable CORS**.
3. Check the top two boxes for **DEFAULT 4XX** and **DEFAULT 5XX**.
4. Leave the other settings as they are. Choose **Enable CORS and replace existing CORS headers**.
5. IF you see "replace existing values" choose **Yes, replace existing values**. (Ignore any crosses and warnings).

**Now your API is CORS enabled, you are ready to deploy your API.**

1. Choose **Actions**. Under **API Actions**, select **Deploy API**.
2. On the **Deploy API** pop-up:
3. For **Deployment stage**, select **[New Stage]**
4. For **Stage name**, enter prod (lowercase)
5. For **Stage description**, enter prod
6. Leave **Deployment description** blank.
7. Choose **Deploy**. (Ignore any warnings).

You should now be provided with a URL in your DragonSearch API dashboard that looks like this:

https://xxxxxxxxxxxxx.execute-api.us-east-1.amazonaws.com/prod

* Copy this link to your clipboard.

You now need to edit your website configuration config.js file to point to your new API endpoint.

## Step 5: Update a config file to point to new API (mock) endpoint

Head back to Cloud 9, and close the scan\_dragons.js tab as you no longer need it open.

Currently, your website has the following in a config.js file in the resources/website folder. Double click it to edit it.

It will show this:

var API\_ENDPOINT\_STR = "<FMI>";

You need to edit this file to point to your new API endpoint:

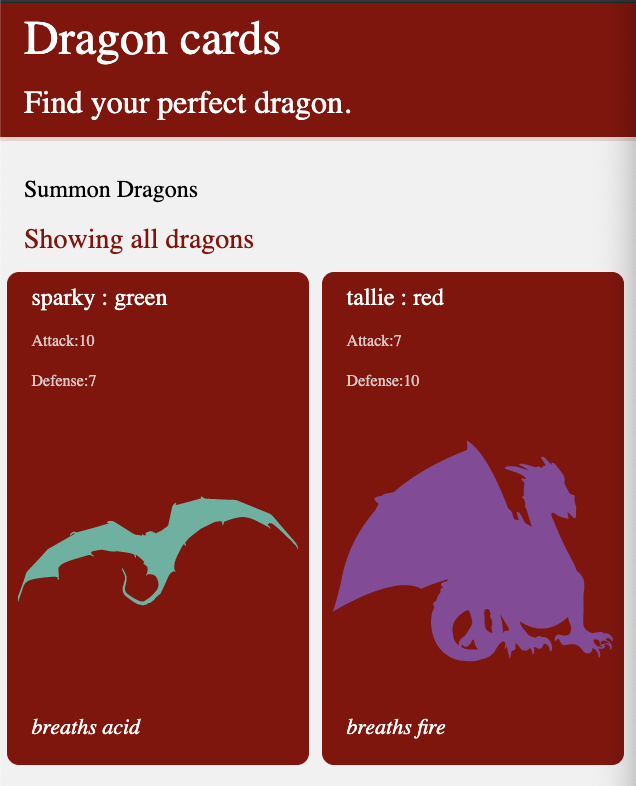
1. Replace the with your API endpoint in quotes, like this: Remove any trailing slash after prod (if you have one), like so:
2. var API\_ENDPOINT\_STR = "https://xxxxxxxxxxxxx.execute-api.us-east-1.amazonaws.com/prod"
3. Save this config.js file and close that tab
4. Open resources/upload\_config.js and then **swap the for your bucket**. Then issue this this command in the Cloud9 terminal.
5. pushd /home/ec2-user/environment/lab2/resources && node upload\_config.js && popd

You should see something a bit like this if it worked:

null { ETag: '"1b3c39cfa4c51b2d148635300d082c4d"' }

Now navigate back to your s3 website, and press refresh.

You should see both dragons.



Now all the website plumbing is out of the way (phew!). You have a simple proof of concept you can show Mary.

Hopefully she will provide you with real dragon data soon, so that you can adjust your function to query on specific dragons by name.

You are also crossing your fingers that she gives you a card template concept, and more dragon images, as she only gave you these two.

**Congratulations!** You have completed Exercise 2.

## Exercise 3

[version\_1.0.1]

# Exercise: Adding multiple items to Amazon DynamoDB using the AWS Software Development Kit (AWS SDK)

In this exercise, you will learn how to develop with Amazon DynamoDB by using the AWS Software Development Kit (AWS SDK). Following the scenario provided, you will add multiple items into multiple DynamoDB tables using the AWS SDK. This exercise gives you hands-on experience with both Amazon DynamoDB and AWS Cloud9.

## Objectives

After completing this exercise, you will be able to use the AWS SDKs to do the following:

* Uploading multiple items to multiple DynamoDB tables.

## Story continued

So you now have a proof of concept website that on page load will display dragon cards (albeit 2) using data stored in DynamoDB.

Your website communicates with your API gateway backend, and currently returns everything in the database.

This is fine for a proof of concept, and when you call Mary and show her the website she is very happy with what you are doing. She tells you that she already has come up with an idea for a card template and asks if you can integrate that too?

You say no problem, and ask her to email it to you along with the JSON files for all the dragon data that she has been promising you all week.

She apologizes about the delay and tells you, you will have it in the next 5 minutes via email.

38 minutes later, you get the email. Your head sinks into your hands, as you see that she has a much more complex data requirement than the one you envisioned.

It is basically a relational data structure with dragons of different types, having different skills and modifiers. Essentially she has given you all the data that would be required for an actual game engine.

You are not building the game engine. She has other people lined up for that, thank goodness. However she still wants you to use DynamoDB to store this data for her, and leverage that to display card data on the website.

You already know how to upload one item at a time with code, but that isn't feasible with the amount of data she has given you.

You need to come up with a script that can upload multiple items to multiple tables (using batch processing).

The 4 JSON files she sent you have the following structure:

###### DragonStatsTableOne [ 25 items]

[{

   "damage\_int": 9,

   "description\_str": "From the northern fire tribe, Atlas was born from the ashes of his fallen father in combat. He is fearless and does not fear battle.",

   "dragon\_name\_str": "Atlas",

   "family\_str": "red",

   "location\_city\_str": "anchorage",

   "location\_country\_str": "usa",

   "location\_neighborhood\_str": "w fireweed ln",

   "location\_state\_str": "alaska",

   "protection\_int": 7

}...]

###### DragonStatsTableTwo [25 items]

(as above) - she just broke them into manageable files, rather than use one big file.

###### DragonCurrentPowerTable [1 item] - sample item only (table is dynamic)

[{

"current\_endurance\_int": 3,

"current\_will\_not\_fight\_credits\_int": 2,

"dragon\_name\_str": "Cassidiuma",

"game\_id\_str": "56syjdh8756"

}...]

###### DragonBonusAttackTable [5 items]

[{

"breath\_attack\_str": "acid",

"description\_str": "spews acid",

"extra\_damage\_int": 3,

"range\_int": 5

}...]

###### DragonFamilyTable [4 items]

{

"breath\_attack\_str": "acid",

"damage\_modifier\_int": -2,

"description\_str": "Better defense",

"family\_str": "green",

"protection\_modifier\_int": 2

}...]

You decide to create a table for each JSON file, and structure it a bit like this:

## Dragon Stats Table

You decide to use dragon\_name as the Primary Key (PK) as you will want to search for a dragon by name.

| **dragon\_name (PK)** | **damage** | **description** | **protection** | **family** | **location\_country** | **location\_city** | **location\_state** | **Location\_neighborhood** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Atlas | 9 | From the northern fire tribe, Atlas was born from the ashes of his fallen father in combat. He is fearless and does not fear battle. | 7 | red | usa | anchorage | alaska | w fireweed ln |

## Dragon current power table [ 1 per game]

Mary tells you that these entries will be created dynamically per game, and she has given you an example.

Although you won't be referring this data in your card viewing website, she gave it you anyway for context.

You decide that the string game\_id (String) should probably be the primary key (PK). They will likely want to search on game\_id and get a list of all the dragons in play.

| **game\_id (PK)** | **dragon\_name** | **current\_will\_not\_fight\_credits** | **current\_endurance (dynamic)** |
| --- | --- | --- | --- |
| 56syjdh8756 | cassidiuma | 2 | 3 |

## Dragon Bonus Attack Table

As you think that they might want to search for details on a type of breath\_attack you choose that as your primary key for this table.

Also you think it would be nice to see if an attack is "in range", so you add a sort key on range to find out if say "a water attack is in range to do damage".

| **breath\_attack(PK)** | **description** | **extra\_damage** | **range (SK)** |
| --- | --- | --- | --- |
| acid | spews acid | 3 | 5 |
| electricty | bolts fly from mouth | 3 | 5 |
| water | high pressure jet over a large area | 1 | 10 |
| fear | Prevent all attacks next round | 0 | 4 |
| fire | Short blast of fire | 8 | 4 |

## Dragon Family Table

Later on in the game engine, the developers will likely need to be able to bring up information about modifiers relating to the dragon type (family), so you decide to use family as the Primary Key (PK).

You feel that this table should probably be merged with the bonus attack table, but you keep it the way it is for now. [You can always optimize this later on].

| **breath\_attack** | **damage\_modifer** | **description** | **family (PK)** | **protection\_modifier** |
| --- | --- | --- | --- | --- |
| acid | -2 | Better defense | green | 2 |
| fire | 2 | Attacks faster | red | -2 |
| water | 1 | Happy in water | blue | -1 |
| fear | 0 | Prefers to bite | black | 0 |

## Time to load data

You remember reading about different ways to get data into DynamoDB, and you figure that using the SDK and the **batchWriteItem** method would be the best option here.

## Prepare the exercise

Before you can start this exercise, you need to import some files and install some modules in the AWS Cloud9 environment that has been prepared for you.

1. Head over to Cloud9.
2. As you have been working with Cloud9 for the previous exercises we need to make sure you are in the right path in the thermal.

cd ~/environment

You should probably close any Cloud9 tabs you have open, and collapse any inactive folders, as it can get messy with too many code tabs open.

1. You will need to seed your Cloud9 filesystem with the lab3 content, so go to the Cloud 9 **bash terminal** (at the bottom of the page) and run the following wget command:

wget https://s3.amazonaws.com/awsu-hosting/edx\_dynamo/c9/dynamo-seed/lab3.zip -P /home/ec2-user/environment

1. To unzip the workdir.zip file, run the following command:

unzip lab3.zip

1. To keep things clean, run the following commands to remove the zip file:

rm lab3.zip

1. Select the black arrow next to the lab3 folder to expand it (collapsing lab 2 and 1 if they are still open). Notice that there is a solution folder in lab3 along with a resources folder. Throughout this exercise, **don't peek at the solution unless you really get stuck. Always TRY to code first.** We will come back to the resources folder shortly. For now let's set the terminal path to the correct folder.
2. cd lab3

??

As this course is self paced, often people will start the lab then come back to it later. In the interim period we may have made adjustments to the code.

Ensure that inside your *lab3* folder that the name of the version markdown file is matching the version number at the top of this document.

If they are out of sync, you will run into problems, to get them synced simply remove the old folder, ensure you are in the right path in the terminal, and run through the wget steps above one more time,

This lab and future labs are especially prone to getting out of sync, because all the website iterations have alredy been loaded into lab2.

So **if** your website is not working as expected for the lab, please double check that **lab2** (that contains the website) has not had any updates recently. If your lab 2 code versions doesn't match the version on the lab2 document, you will need to re-upload the website by following the steps one more time from **lab 2**.

You are now ready to do the exercise tasks.

#### Step 1A): Create multiple DynamoDB tables using the SDK

You already know how to create tables using the AWS-SDK, so creating a few more should be a breeze. Follow these steps, to create the following tables with respective Primary Keys (see tables above).

Just replace the 's in the following script.

1. Open up lab3/create\_multiple\_tables.js and edit as needed.
2. Once you are done choose File and Save.
3. Run the following:
4. node create\_multiple\_tables.js

You should see output like the following:

[ { TableDescription:

{ AttributeDefinitions: [Array],

TableName: 'dragon\_stats',

KeySchema: [Array],

TableStatus: 'CREATING',

CreationDateTime: 2019-05-17T18:43:31.060Z,

ProvisionedThroughput: [Object],

TableSizeBytes: 0,

ItemCount: 0,

TableArn: 'arn:aws:dynamodb:us-east-1:<000000000000>:table/dragon\_stats',

TableId: '6bde2328-7075-4f28-b5f7-7f21df30df36',

BillingModeSummary: [Object] } },

{ TableDescription:

{ AttributeDefinitions: [Array],

TableName: 'dragon\_current\_power',

KeySchema: [Array],

TableStatus: 'CREATING',

CreationDateTime: 2019-05-17T18:43:31.015Z,

ProvisionedThroughput: [Object],

TableSizeBytes: 0,

ItemCount: 0,

TableArn: 'arn:aws:dynamodb:us-east-1:<000000000000>:table/dragon\_current\_power',

TableId: 'b8b7dbea-fa97-4f21-a59c-99210fd2192f',

BillingModeSummary: [Object] } },

{ TableDescription:

{ AttributeDefinitions: [Array],

TableName: 'dragon\_bonus\_attack',

KeySchema: [Array],

TableStatus: 'CREATING',

CreationDateTime: 2019-05-17T18:43:31.050Z,

ProvisionedThroughput: [Object],

TableSizeBytes: 0,

ItemCount: 0,

TableArn: 'arn:aws:dynamodb:us-east-1:<000000000000>:table/dragon\_bonus\_attack',

TableId: 'aff78f2f-990c-432e-bd90-9eb0c00939aa',

BillingModeSummary: [Object] } },

{ TableDescription:

{ AttributeDefinitions: [Array],

TableName: 'dragon\_family',

KeySchema: [Array],

TableStatus: 'CREATING',

CreationDateTime: 2019-05-17T18:43:31.018Z,

ProvisionedThroughput: [Object],

TableSizeBytes: 0,

ItemCount: 0,

TableArn: 'arn:aws:dynamodb:us-east-1:<000000000000>:table/dragon\_family',

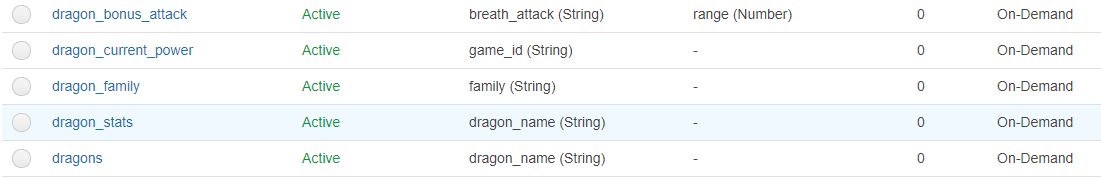
TableId: '46498747-903c-4a11-b551-4207649c1930',

BillingModeSummary: [Object] } } ]

HowFastWasThat: 133.393ms

Note: the only change in terms of code vs what you did last time in lab 1, is that we are using the Async Await feature of Node to allow us to use Promises. Don't get hung up on this stuff, it's just a way to carry out multiple calls to the DB in one script.

Also you may have to wait up to f**ive minutes** before all your tables show as **ACTIVE**.



Make sure your tables say ***Active*** before moving on. Adding items when in the "creating" state will bork!

#### Step 1B): Add Dragon data to multiple DynamoDB tables using the BatchWriteItem method (SDK)

1. Open the SDK docs for now to find the method for creating new items as a batch inside an existing DynamoDB table. Find out the correct method name and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#batchWriteItem-property> |

**Time to write some code that adds a a batch of items to each of the DynamoDB tables.**

1. Open up the seed\_dragons file inside lab 2 by double clicking on it.
2. Have the SDK docs open (as above) to help you.
3. Replace the sections of the code in that file, so that the code uploads all the dragon data into new tables. You table should have been called **dragons** and will be in **us-east-1**.
4. You have the dragon data in the JSON files.
5. Save the file.
6. Double check your code is correct by looking at the solution file.
7. Go to the terminal and run your file
8. node seed\_dragons.js

**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

## Confirm that your code worked.

You should see the following out after running:

$ node seed\_dragons.js

[ { UnprocessedItems: {} },

{ UnprocessedItems: {} },

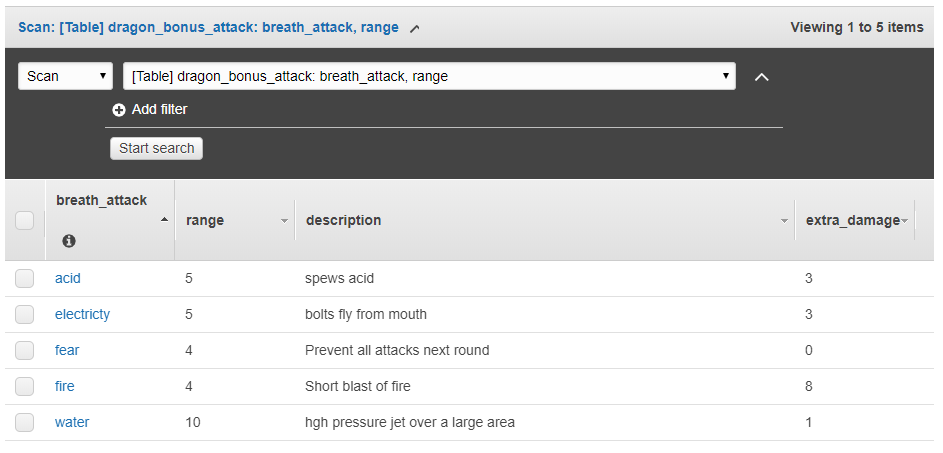
{ UnprocessedItems: {} },

{ UnprocessedItems: {} },

{ UnprocessedItems: {} } ]

HowFastWasThat: 183.082ms

We can also verify in the DynamoDB console refresh the page, and look at the **items** tab of one of the tables. In this case let's look at the dragon\_bonus\_attack table.



Awesome. We have all the dragon data in the database. The challenge is that the code you wrote earlier that scans the table (remember lab2) was written to accommodate a different table name **dragons**, and has a completely different schema..

#### Step 1C): Delete the old table

We are no longer going to use the mock dragon database you created in lab1. So this would be a great time to clean it up.

Run this command from the Cloud 9 terminal to remove the old dragon table. Out with the old in with the new.

aws dynamodb delete-table --table-name dragons

You should see something like this:

{

"TableDescription": {

"TableArn": "arn:aws:dynamodb:us-east-1:<000000000000>:table/dragons",

"ProvisionedThroughput": {

"NumberOfDecreasesToday": 0,

"WriteCapacityUnits": 0,

"ReadCapacityUnits": 0

},

"TableSizeBytes": 0,

"TableName": "dragons",

"BillingModeSummary": {

"LastUpdateToPayPerRequestDateTime": 1558029188.272,

"BillingMode": "PAY\_PER\_REQUEST"

},

"TableStatus": "DELETING",

"TableId": "78e4aeb0-c70e-4678-bbee-60e97f7f3524",

"ItemCount": 0

}

}

Now our website will break because the API is pointing to a Lambda function that references the table you just removed.

Fear not. All we need to do is edit our Lambda function to point to the right table.

Earlier you updated your website to allow a user to search for a specific dragon and bring up just that dragon. They can still select "show all dragons", but you thought it was a nice feature so you added it.

So while we are in there updating the Lambda function, we will have you update the code a little more, to allow for more functionality.

The website needs to show a single dragon card If the website passes a dragon name in the request. If they don't choose a dragon we just give them a table dump ;)

## Step 2: Adjust the lambda function to work with the new database, and filter for just a specific dragon (if asked)

So your current API function takes in an empty POST request and spits out everything in the dragon data table.

You will need to modify lab3/scan\_dragons.js to allow a user to search for a dragon by name or search for all dragons!

### CODE STEPS:

The solution can be found in the solution folder if you get stuck.

You "could" edit the code directly in the Lambda function, but you would like to test it in Cloud 9 first. Then you can copy and paste it into Lambda.

You want to be able to test a "scan all", and a show a specific dragon.

Open the SDK docs and find the method for scanning and filtering with DynamoDB. Find the correct method name and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#scan-property> |

**Time to write some code that adds a a batch of items to each of the DynamoDB tables.**

1. Open up the scan\_dragons file double clicking on it. (note use the scan\_dragons.js file inside the lab3 folder **not** the old lab2 folder)
2. Collapse and close and files you are not using.
3. Have the SDK docs open (as above) to help you.
4. Replace the sections of the code in that file, so that the code scans all the dragon data from the dragon\_scan table.
5. Run the following:
6. node scan\_dragons.js test "Cassidiuma"
7. You should see the following:
8. Local test for a dragon called Cassidiuma
9. null [ { location\_neighborhood: { S: 'poplar st' },
10. damage: { N: '7' },
11. location\_city: { S: 'colby' },
12. family: { S: 'green' },
13. description:
14. { S: 'Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen\'s most loved and feared warrior.' },
15. protection: { N: '10' },
16. location\_country: { S: 'usa' },
17. location\_state: { S: 'kansas' },
18. dragon\_name: { S: 'Cassidiuma' } } ]
19. Run the following:
20. node scan\_dragons.js test

It should return ALL the dragons in the table.

## Step 3: Copy and paste new code into the lambda console (updating only Lambda and test steps)

1. Pivot back to the console by choosing **AWS Cloud9** in the upper left.
2. Choose **Go To Your Dashboard** and choose **Services** and search for **Lambda**.
3. Open the **DragonSearch** function.
4. Replace the contents of index.js with the new code from scan\_dragons.
5. Choose **Save**.
6. Choose **Test** (using the **DragonScan** test event).
7. It should show you a list of dragons in the **Details** section under Execution results.
8. Now try searching for a dragon by name.
9. Click the drop down area next to **Test** and choose **Configure test events**.
10. Select the radio button next to **Create new test event**.
11. Under **Event name** enter JustOneDragon.
12. Replace the existing content with:
13. {
14. "dragon\_name\_str": "Cassidiuma"
15. }
16. Choose **Create** and then choose **Test**.
17. In the **Execution results** you should see the following:
18. [
19. {
20. "location\_neighborhood": {
21. "S": "poplar st"
22. },
23. "damage": {
24. "N": "7"
25. },
26. "location\_city": {
27. "S": "colby"
28. },
29. "family": {
30. "S": "green"
31. },
32. "description": {
33. "S": "Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen's most loved and feared warrior."
34. },
35. "protection": {
36. "N": "10"
37. },
38. "location\_country": {
39. "S": "usa"
40. },
41. "location\_state": {
42. "S": "kansas"
43. },
44. "dragon\_name": {
45. "S": "Cassidiuma"
46. }
47. }
48. ]

## Step 4: Test the existing REST API endpoint and connect it to your website

Good news the API you set up in lab2 is pointing to the ARN (Amazon Resource Name) of that Lambda function you just edited.

This means you do not need to re-deploy your API gateway. However a quick test before checking the website is best practice. So follow these steps.

1. Choose **Services** and search for **API Gateway**.
2. Choose the **DragonSearchAPI**.
3. Choose **POST** and select **TEST**.

You can leave the "request body" blank or use "All" shown below:

{

"dragon\_name\_str": "All"

}

1. Choose **Test**.

You will see all the dragon information returned at the right.

Now let's test for a specific dragon.

1. In the **Request Body** paste the following:
2. {
3. "dragon\_name\_str": "Cassidiuma"
4. }
5. Choose **Test**.
6. You should see the following in the **Response Body**
7. [
8. {
9. "location\_neighborhood": {
10. "S": "poplar st"
11. },
12. "damage": {
13. "N": "7"
14. },
15. "location\_city": {
16. "S": "colby"
17. },
18. "family": {
19. "S": "green"
20. },
21. "description": {
22. "S": "Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen's most loved and feared warrior."
23. },
24. "protection": {
25. "N": "10"
26. },
27. "location\_country": {
28. "S": "usa"
29. },
30. "location\_state": {
31. "S": "kansas"
32. },
33. "dragon\_name": {
34. "S": "Cassidiuma"
35. }
36. }
37. ]

**Great news** your web back-end API is all good to go. However your old website would be expecting different data and now no longer work.

Luckily you had some time over the weekend to update your site and push it to S3.

So you can test the newer version 2.0 (index2.html) of your web front end.

NOTE: This was already uploaded when you did lab 2, you just need to navigate to *index2.html*.

If you don't have a tab with your website open in chrome, do the following to find that url.

1. Choose **Services** and search for **s3**.
2. Choose the S3 bucket that was created in the previous lab: 2019-05-16-dynamolab-er-102
3. Choose **Properties** and **Static website hosting**.

Once you have the url, add index2.html to the end of it like so:

1. Open the Endpoint in a new browser tab and append /index2.html to it:
2. http://<your s3 bucket>.s3-website-us-east-1.amazonaws.com/index2.html

You should see all the dragins, AND be able to choose just one dragon!

**Congrats** Lab 3 completed, you have lots of Dragon data inside multiple tables in DynamoDB wired up to your website.

AWESOME! Mary will be happy!

## Exercise 4

[version 1.0.3]

# Exercise: Monitoring Amazon DynamoDB using the AWS Software Development Kit (AWS SDK)

## Overview

In this exercise, you will learn how to develop with Amazon DynamoDB by using the AWS Software Development Kit (AWS SDK). Following the scenario provided, you will learn how to monitor and debug your DynamoDB backed application, using CloudWatch Metrics, CloudWatch Alarms, CloudWatch Logs, and AWS X-Ray. This exercise gives you hands-on experience with Amazon CloudWatch and AWS X-Ray.

## Objectives

After completing this exercise, you will be able to use Amazon CloudWatch and AWS X-Ray to do the following:

* View CloudWatch Metrics
* Create CloudWatch Alarms
* View and search CloudWatch Logs
* Use the X-Ray Service Map to detect issues
* Use the X-Ray Traces to dive into issues

## Story continued

You are happy that you were able to get a proof of concept over to Mary. However last night Mary's project manager (Steve) decided to make some adjustments to your code. The reason for the code change is to add pagination to the scan getting the entire list of dragons, as the table will grow over time. He heard that results are divided into "pages" of data that are 1 MB in size (or less). This means that the application needs to process the first page of results, then the second page, and so on.

You are happy that he's doing this code change for you as it's not something you've done in the past. He also told you that he would instrument the code with X-Ray as he just followed a class on this. Meanwhile you can work on other projects and you will be able to learn how he did those things by reading the code.

You know Steve, and he's a great guy but has created issues with his code in the past. He really doesn't like testing before deploying new code. This means that the probability of having something that breaks in production is high.

Since Continuous Integration hasn't been implemented yet in this Development environment, you decide to set an alarm in CloudWatch to notify yourself, just in case anything breaks.

Another issue that you have been working on is performance optimization. You have heard of the difference between a Scan with a Filter and a Query, but would like to see a proof of the difference. So you decided to instrument the code with X-Ray to see the difference in latency.

## Prepare the exercise

Before you can start this exercise, you need to import some files in the AWS Cloud9 environment.

1. From the AWS Management Console, go to the **Services** menu and choose **Cloud9**.
2. Choose **Open IDE** to open the AWS Cloud9 environment.
3. Ensure you are in the root directory, and close down and tabs and collapse any folders you are not using.
4. cd /home/ec2-user/environment
5. To get the files that will be used for this lab, go to the Cloud 9 **bash terminal** (at the bottom of the page) and run the following wget command:

wget https://s3.amazonaws.com/awsu-hosting/edx\_dynamo/c9/dynamo-monitor/lab4.zip -P /home/ec2-user/environment

You should also see that a root folder called **dynamolab** with a lab4.zip file has been downloaded and added to your AWS Cloud9 filesystem (on the top left).

1. To unzip the lab4.zip file, run the following command:

unzip lab4.zip

In your Cloud9 filesystem, you should see a lab4 folder.

1. To keep things clean, run the following commands to remove the zip:

rm lab4.zip

cd lab4

echo "done"

1. Once you see "done", select the black arrow next to the lab4 folder (top left) to expand it. Notice inside this lab4 folder there is a solution folder. **Try not to peek at the solution unless you really get stuck. Always TRY to code first.**

Before you look at Steve's code, you think the most important thing to do is activate full monitoring in case something breaks, and if it does break, be able to find out why it broke using these gathered metrics.

## Exercise 1: Activate X-Ray in API Gateway

AWS X-Ray is a great way to monitor how data is moving through your application (think x-ray vision)

To understand the whole data flow, you will activate X-Ray in API Gateway. When API Gateway communicates with Lambda, it will use the context to let Lambda know that X-Ray was activated. Lambda will automatically enable it for that call.

By activating X-Ray in API Gateway, you will see the traffic flow to your Lambda function, but not all the way over to DynamoDB. To be able to see a call made to DynamoDB, the code itself will need to be instrumented with the X-Ray SDK. Let's enable things at the API Gateway part first.

1. From **Cloud9**, click the **AWS Cloud9** button next to File.
2. Select **Go To Your Dashboard**.
3. Choose the **Services** menu and choose **API Gateway**.
4. Choose the **DragonSearchAPI** link on the left menu.
5. Choose the **Stages** link on the left menu.
6. Choose the **prod** link.
7. Choose the **Logs/Tracing** tab.
8. Under **X-Ray Tracing**, put a check mark next to **Enable X-Ray Tracing**.
9. Choose the **Save Changes** button.
10. Then redeploy the API. **Actions** > **Deploy** (prod)

Now we have X-Ray tracing enabled, we should head over to Amazon CloudWatch and look at the current information we have about the prior interactions with DynamoDB.. - optional step (recommended)

## Step 2: Explore the CloudWatch Metrics of DynamoDB -[optional section]

1. Choose the **Services** menu and search for **CloudWatch**.
2. Click **Metrics** in the left menu.
3. Click **DynamoDB** in the **All metrics** tab.
4. Under **Table Metrics**, you will find the amount of consumed RCU and WCU for the calls you have made on your table in the previous exercises. By default, the time period is set to 3 hours, so you may have to set the time period to longer depending on when you did the previous exercises (see top of console).
5. Feel free to explore the other metrics that are available. Note that you will only see the metrics that have data. You can find the list of CloudWatch Metric Dimensions for DynamoDB here:

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/metrics-dimensions.html

The goal of this task is for you to simply explore the console of CloudWatch and understand that there are different metrics.

⚠️ **Command line alert**

To mix things up a bit, and keep things interesting for you during these labs, we are going to have you complete this particular lab using the command line only.

Yep. No code, no 's', no scripting.

Just for a change!

I think it is valuable for even you coders to learn the basics of the CLI. You don't abve to be a systems admin to do this ;), just roll with it it's pretty straight forward.

TIP Copy these commands to a text editor first before editing them (or use a new file in your Cloud 9 environment) so editing the commands that use below are a bit easier to work with.

## Step 3: Get alerted on UserErrors

In this task, you will create a **CloudWatch Alarm** from based on the metric **UserErrors**. The alarm will notify a Simple Notification Service Topic that you will create and that you will subscribe to using an email address that you have access to.

1. In the **Cloud 9 terminal**, execute the following command to create a Simple Notification Service Topic.

aws sns create-topic --name edx-ddb-monitor

The result will look similar to the following. It gives you the SNS Topic ARN that will be used in the next steps.

{

   "TopicArn": "arn:aws:sns:us-east-1:xxxxxxxxxxx:edx-ddb-monitor"

}

Each time the SNS Topic ARN is referred in the next step, the value that you must use is arn:aws:sns:us-east-1:xxxxxxxx:edx-ddb-monitor based on the result above. Your ARN is different, make sure that you use your ARN and not the one we gave you here.

1. Subscribe your email address to the SNS Topic you created by running the following command. Make sure you replace <SNS TOPIC ARN> with the SNS Topic ARN you received in the previous step and <YOUR EMAIL ADDRESS> with an email address you have access to as you will receive an email to confirm.

aws sns subscribe --topic-arn <SNS TOPIC ARN> --protocol email --notification-endpoint <YOUR EMAIL ADDRESS>

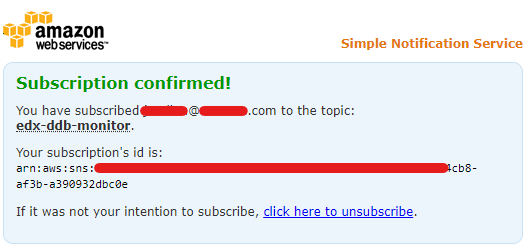
The result will look like the following:

{

"SubscriptionArn": "pending confirmation"

}

1. You will receive an email at the address that you provided. It will be from [**no-reply@sns.amazonaws.com**](mailto:no-reply@sns.amazonaws.com) and will contain a link to confirm your subscription. In that email, click on the link **Confirm subscription**. The email will be similar to the following:
2. You have chosen to subscribe to the topic:
3. arn:aws:sns:us-east-1:123456789012:edx-ddb-monitor
4. To confirm this subscription, click or visit the link below (If this was in error no action is necessary):
5. Confirm subscription
6. Please do not reply directly to this email. If you wish to remove yourself from receiving all future SNS subscription confirmation requests please send an email to sns-opt-out

A web page will open that will look like the following: 

1. To create the Amazon CloudWatch Alarm, execute the following command in the **Cloud 9terminal**. Replace <SNS TOPIC ARN> with the SNS Topic **ARN** that you created in a previous step.

This command takes many parameters:

* + alarm-name: The name of the alarm.
  + alarm-description: The description for the alarm.
  + namespace: The namespace for the metric associated specified in metric-name. In this case, this is under DynamoDB.
  + metric-name: The name for the metric associated with the alarm. Errors that are generated by your code, would increase the UserErrors metric.
  + statistic: The statistic for the metric specified. We are using Sum for this period of time.
  + period: The length, in seconds, used each time the metric specified is evaluated.
  + evaluation-periods: The number of periods over which data is compared to the specified threshold. In this case, we want to trigger if we see anything above 0 in the last 60 seconds, so it's set to 1.
  + threshold: The value against which the specified statistic is compared. As we want to trigger any time the metric goes above 0, it's set it to 0.
  + comparison-operator: The arithmetic operation to use when comparing the specified statistic and threshold. In this case, it's set to GreaterThanThreshold meaning each time it's above 0 in this specific case.
  + unit: The unit of measure for the statistic. It's a Count in this case.
  + alarm-actions: The actions to execute when this alarm transitions to the ALARM state from any other state. In this case, the goal is to trigger your SNS Topic.

aws cloudwatch put-metric-alarm --alarm-name DDB-UserErrors --alarm-description "Alarm when UserErrors in DynamoDB exceeds 0" --namespace AWS/DynamoDB --metric-name UserErrors --statistic Sum --period 60 --evaluation-periods 1 --threshold 0 --comparison-operator GreaterThanThreshold --unit Count --alarm-actions <SNS TOPIC ARN>

If the command worked, you shouldn't receive any errors nor output (no news is good news).

Lets refelct on what we just did there.

We a) emnabled Xray for Apigateway so we can track future requests comomng form the website, and B) we have set an alarm (the Steve alarm) to make sure that is something breaks and we get errors we find out sooner rather than later.

Before you move on to step to you drop Steve an amazon link to a kindle book on Test Driiven Code to wind him up ;).

## Step 4: Deploy Steve's code and simulate a user

Since Steve doesn't have access to your AWS Account, you are going to uplaod the code he emialed you and deploy it (unchecfked).

You hate yourslef for pushing (untested) code and overwriting you old "tested" code, but you brace yourself and consider that the learning experience in fixing anything that happens to go wrong is worth it. Besides its only you and Mary's team that are seeing this thing anyway at this point.

So the next step is to blindly deploy his new version of the Lambda function using a zip file that he gave you (/lab4/resources/steve-code.zip). One uploaded you should hit the website a few times and do a few searches on it to generate some metrics and trigger all the CloudWatch and XRAY stuff you jiust stet up.

1. To overwrite your tested code with steves , execute the following CLI command from the **cloud 9 terminal**/
2. cd ~/environment/lab4
3. aws lambda update-function-code --function-name DragonSearch --zip-file fileb://resources/steve-code.zip

You should see something like this:

{

"FunctionName": "DragonSearch",

"LastModified": "2019-06-07T21:03:52.177+0000",

"RevisionId": "84e123bb-2269-47fd-a3ab-381f52f2c8cc",

"MemorySize": 128,

"Version": "$LATEST",

"Role": "arn:aws:iam::xxxxxxxxxx:role/call-dynamodb-role",

"Timeout": 10,

"Runtime": "nodejs10.x",

"TracingConfig": {

"Mode": "PassThrough"

},

"CodeSha256": "0DUTwv1A2BkXFqvDKdD03s2M5+l4v5JHfcSViRyKs4s=",

"Description": "",

"VpcConfig": {

"SubnetIds": [],

"VpcId": "",

"SecurityGroupIds": []

},

"CodeSize": 1567593,

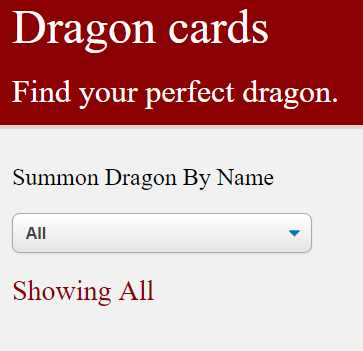
"FunctionArn": "arn:aws:lambda:us-east-1:xxxxxxxx:function:DragonSearch",

"Handler": "index.handler"

}

1. Head over to the current latest version of your website (index2.html). **REFRESH**

You should see the following:



Wait. shouldn't there be a list of dragons here like the previous exercise, something is definitely wrong, you know there are dragins as you have seen them before. Somits-rong

1. Press refresh a few times.
2. Now do a search for Fireball.
3. Then do a a search for Dexlar.
4. Then select All

..YEP All is broken.

You remember that hte All triggers the SCAN function that Steve said he wanted to paginate, right?

Time to find out what he did in the code that is breaking it!

**FYI In the next 10 minutes, you will receive an alarm letting you know that an error has occurred.**

As you already found out there is a breaking error, you can jump onto this straight away and find out what is going wrong.

## Step 5: Find the issue (that you already know is in Steve's code)

Hopefuly you have just received an alarm via an email that a **UserErrors** on a DynamoDB table just happened. If not, it will come through soon.

As per the AWS docs:

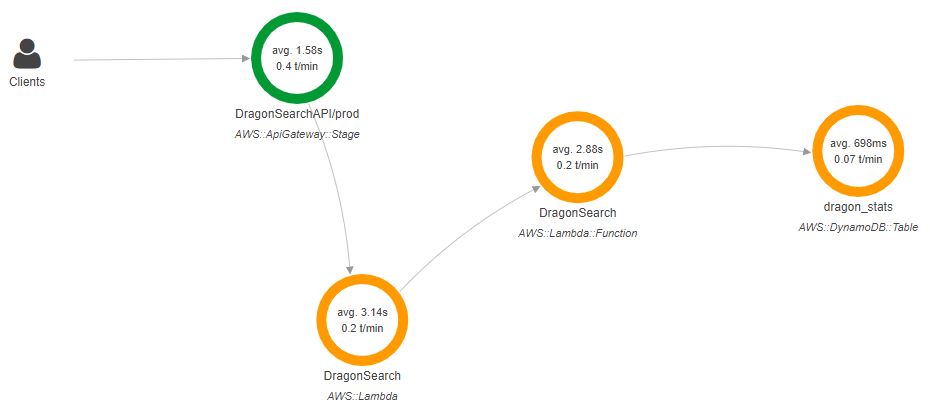
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/metrics-dimensions.html

UserErrors means that a request to DynamoDB generated a 400 status code. This means that there is a problem with the request sent. You know the error is coming from the Lambda function where Steve did a code change. However, in an environment where Steve could have done code changes in many different places (other Lambda or API Gateway), it can become difficult to determine where the error is coming from. You could look in the logs of API Gateway and Lambda, but if this was a complex environment with many microservices, it would be much more difficult to trace back. One of the services to help map this, is X-Ray.

Thankfully Steve instrumented his new code with X-Ray. This is awseome and you are pleased he did this, becuase it menas thaty not only cna you see the PAI gawatey Xray stuff but all ws what is happeoimng inside the code at the DynamoDBsection.

Instumenting with AWS XRAY is easy BTW. When you view Steve's code (which you will do in a bit) you will see how easy it was to implement.

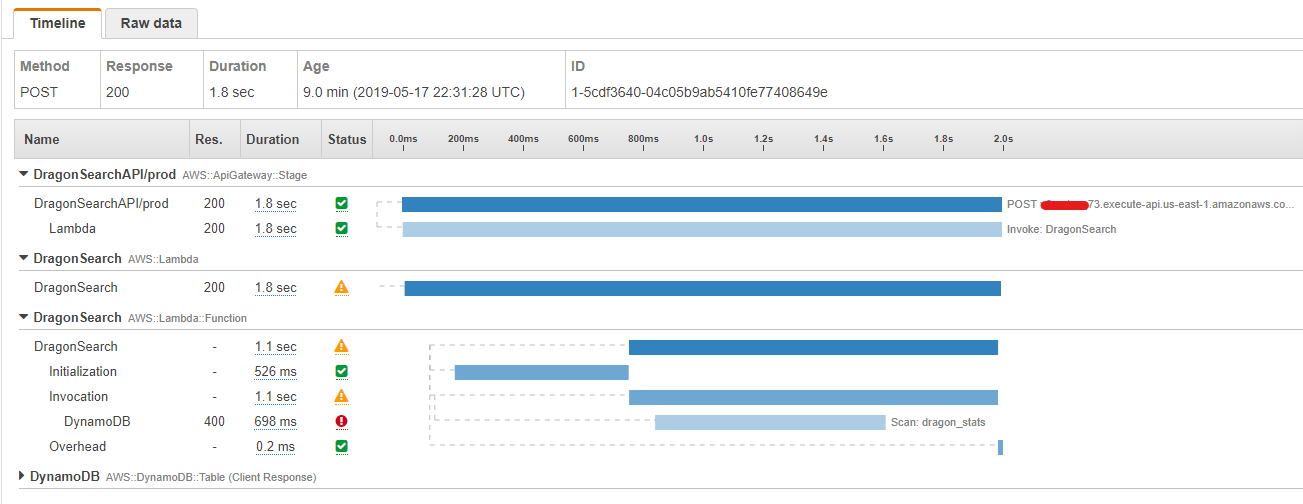
###### **Let's go troubleshooting. staring with XRAY**

1. Click the **Services** menu and choose **X-Ray**.
2. If this is the first time you go to the X-Ray console, you will be presented with a welcome screen. Click **Get started** and click **Cancel**.
3. Then in the left menu, click on **Service map**.
4. You will see a map similar to the following. If you don't see that map, note that X-Ray defaults to the Last 5 minutes. If you have executed the test from the last tasks more than 5 minutes ago, click on the **Last 5 minutes** dropdown and select 15 or 30 depending on your use case. As you can see the client (you and your Chrome browser) sent a request to DragonSearchAPI in the prod stage. That request was **successful** as it's represented in green. You can click on the Map legend link on the right to know what the colors mean. Yellow means there was a **4xx** error. Notice that the trafic then was then sent to AWS Lambda DragonSearch the service which then sent the request to the DragonSearch **function**. This will allow you to see the time of initialization and invocation of your Lambda function. Finally, the code in the Lambda function sent a request to the dragon\_stats DynamoDB Table.

You now have a pretty good understanding of the flow of the application. You now need to dive deeper to know what caused this error. We **all** know it's Steve, but we not here to point blame and we need to fix the code.

1. Click on the **dragon\_stats** in the map which will open a menu on the right.
2. Click on the **View traces >** button.
3. In the **Trace list** table, click on the first ID link. It will start with ... and a sequence of numbers.

This sequence of number is being used by API Gateway, Lambda and the code for every segment being sent. This way, every sub-segment is represented by the same ID so they can all be traced together.

1. You can now see what was described above in the Service Map, but with many more details for this specific Trace. You can see the duration of each section of the call. You can also see the time of Initialization of the Lambda function as this was the first time it was executed. You can also see the type of query sent to DynamoDB and which table (Scan and dragon\_stats) on the call to DynamoDB from within the Lambda function. You can also see the status for each of those calls. One of the status is marked in **red** which probably indicates the error. You can also see the response code sent back which is **400**. This tells us we are on a good path towards finding the error as we have now found which table, what call and what Lambda function caused the error.
2. Expand **DynamoDB** by clicking on the breadcrumb next to it.
3. Click on **DragonSearch** under the DynamoDB section.
4. Click on the **Exceptions** tab.

You can now see the exception that was generated in the call to DynamoDB is a **ValidationException** which is related to a reserved keyword.

Message Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved keyword: family

1. Click the **Close** button.
2. Click on **DynamoDB** under the section **DragonSearch**.

In the Overview tab, you can see the time when this call was made which can help refine your search in the logs if this application was being used by many clients.

1. Click on the **Exceptions** tab.

You can now see the stack trace of the call. Near the bottom of the list, you can see that this exception was generated from the exports.handler function on line 5 which called the scanTable function on line 44. The rest are related to the AWS SDK call.

ValidationException: Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved keyword: family

at features.constructor.captureAWSRequest [as customRequestHandler] (/var/task/node\_modules/aws-xray-sdk-core/lib/patchers/aws\_p.js:83)

at features.constructor.addAllRequestListeners (/var/runtime/node\_modules/aws-sdk/lib/service.js:279)

at features.constructor.makeRequest (/var/runtime/node\_modules/aws-sdk/lib/service.js:203)

at features.constructor.svc.anonymous function [as scan] (/var/runtime/node\_modules/aws-sdk/lib/service.js:673)

at scanTable (/var/task/index.js:44)

at Runtime.exports.handler (/var/task/index.js:5)

at Runtime.handleOnce (/var/runtime/Runtime.js:63)

at process.\_tickCallback (internal/process/next\_tick.js:68)

Note: at scanTable (/var/task/index.js:**44**) and at Runtime.exports.handler (/var/task/index.js:**5**)

1. Click on the **Close** button.
2. Now that you know the issue is in the DragonSearch Lambda function, click the **Services** menu and choose **Lambda**.
3. Click on the Lambda function **DragonSearch**.

**Instead of going right to finding the issue, let's look at the logs of this Lambda function first, just so you are familiar with the Cloudwatch and DynamoDB metrics**

1. Click on the **Monitoring** tab.
2. Click on the button **View logs in CloudWatch**.
3. You can now see a list of Log Streams. There will normally be one log stream per container of your Lambda function. This is why having the time of the event is important in a large environment. Since Mary/you did the test, it's much easier to find as it's the first one at the top of the list. Click on the **first log stream** link.
4. You will see log information for the Lambda function and one of them will contain the error seen previously. Again, knowing the time of the event is of importance, but since you have only executed the code once, there won't be many logs. Another way to find it is by using the search bar. In the **Filter events** search bar, enter ValidationException **as that's the error type we found earlier**.
5. It should return one line, click on that line to **expand it**.

You will then see something similar to this:

2019-06-07T21:20:09.656Z 214675b9-166a-4ee1-b80f-c7ebdbaa9479 ERROR Invoke Error

{

"errorType": "ValidationException",

"errorMessage": "Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved keyword: family",

"code": "ValidationException",

"stack": [

"ValidationException: Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved keyword: family",

" at Request.extractError (/var/runtime/node\_modules/aws-sdk/lib/protocol/json.js:51:27)",

" at Request.callListeners (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:106:20)",

" at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:78:10)",

" at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/request.js:683:14)",

" at Request.transition (/var/runtime/node\_modules/aws-sdk/lib/request.js:22:10)",

" at AcceptorStateMachine.runTo (/var/runtime/node\_modules/aws-sdk/lib/state\_machine.js:14:12)",

" at /var/runtime/node\_modules/aws-sdk/lib/state\_machine.js:26:10",

" at Request.<anonymous> (/var/runtime/node\_modules/aws-sdk/lib/request.js:38:9)",

" at Request.<anonymous> (/var/runtime/node\_modules/aws-sdk/lib/request.js:685:12)",

" at Request.callListeners (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:116:18)"

],

"message": "Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved keyword: family",

"time": "2019-06-07T21:20:09.575Z",

"requestId": "V13L0EJP4IMPV30N8PUMB1LGANVV4KQNSO5AEMVJF66Q9ASUAAJG",

"statusCode": 400,

"retryable": false,

"retryDelay": 40.467102629451915

}

REAL WORLD TIP: So whether you wish to debug using CloudWatch and filtering for message or using XRay it is up to you. I recommend both, like you just did, because sometimes you are not even sure what search terms to use in Cloudwatch. AWS XRAY can help you narrow down the issues to find decent search terms to us in CloudWatch. which then gives you probably more information that you may need ;).CW is especially usuaful if you have added debug statemnets like *console. log*s, because Cloudwatch will show these too, which can be super useful when trouble shooting. In short Xray or CW? The answer is yes

## Step 6: Fix Steve's code

From your investigation from the previous two tasks, you found out that there is an issue in the **scanTable** function in the **DragonSearch** Lambda function which generates the following error:

Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved keyword: family

You also know the line numbers 5 and 44 causing the issue.

The [reserved keyword](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ReservedWords.html) **family** is used for one of the attributes in the dragon\_stats DynamoDB table which is specified in a **ProjectionExpression**.

**It's time to fix some code that makes use of ExpressionAttributeNames to specify reserved keywords in a ProjectionExpression.**

1. Click the **Services** menu and choose **Lambda**.
2. Click on the Lambda function **DragonSearch**.
3. Run a test first justOneDragon

It should be a success, as this uses the query and not the scan

[

{

"location\_neighborhood": {

"S": "poplar st"

},

"damage": {

"N": "7"

},

"location\_city": {

"S": "colby"

},

"family": {

"S": "green"

},

"description": {

"S": "Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen's most loved and feared warrior."

},

"protection": {

"N": "10"

},

"location\_country": {

"S": "usa"

},

"location\_state": {

"S": "kansas"

},

"dragon\_name": {

"S": "Cassidiuma"

}

}

]

So that one is good!

1. Now try the test dragonScan whcih uses the scan, and you should get something like this:

{

"errorType": "ValidationException",

"errorMessage": "Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved keyword: family",

"trace": [

"ValidationException: Invalid ProjectionExpression: Attribute name is a reserved keyword; reserved keyword: family",

" at Request.extractError (/var/runtime/node\_modules/aws-sdk/lib/protocol/json.js:51:27)",

" at Request.callListeners (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:106:20)",

" at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:78:10)",

" at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/request.js:683:14)",

" at Request.transition (/var/runtime/node\_modules/aws-sdk/lib/request.js:22:10)",

" at AcceptorStateMachine.runTo (/var/runtime/node\_modules/aws-sdk/lib/state\_machine.js:14:12)",

" at /var/runtime/node\_modules/aws-sdk/lib/state\_machine.js:26:10",

" at Request.<anonymous> (/var/runtime/node\_modules/aws-sdk/lib/request.js:38:9)",

" at Request.<anonymous> (/var/runtime/node\_modules/aws-sdk/lib/request.js:685:12)",

" at Request.callListeners (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:116:18)"

]

}

1. In the **Function code** section, go to line 44 to find the find the **scanTable** function that is called from line 5. (see how helpful Xray was with those line number #winning)
2. To understand how reserved keywords, ExpressionAttributeNames and ProjectionExpression are all related, visit the AWS documentation:
3. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.ExpressionAttributeNames.html#Expressions.ExpressionAttributeNames.ReservedWords
4. Now that you understand the link, open the node AWS SDK for the scan operatin, and look at how you can modify the **params** variable to make the code work.
5. https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#scan-property
6. You can directly modify the code in the Lambda console, **however** let's use our usual process, test it in Cloud 9 first, then move it to Lambda.
7. Close the lambda console un edited, and head back to Cloud 9
8. Open lab4/steve\_code.js and fix line 44. \*This is all the code you have to do, this lab is 99% CLI and 1% code.
   * If you have issues, you can find the solution of the code in the file *steve-code.js* located in the *lab4/solution* folder of your Cloud9 environment. Replace the entire code in the Lambda function with the code in that file if you get stuck.
9. Before you test the code check your proopsed code with the solution file.
10. Run your code to test the all function which will call the scan method you have just fixed.

Note his conditional installation of XRAY in his code:

# do not paste this in this is just FYI

AWSXRay = require('aws-xray-sdk-core'),\*

AWS = AWSXRay.captureAWS(require('aws-sdk')),

1. Now run the file in the **Cloud9 termina**l.
2. node steve\_code.js Dexler

You should see something like this:

Local test for a dragon called Dexler

null [ { location\_neighborhood: { S: 'bellcastle rd' },

damage: { N: '4' },

location\_city: { S: 'lexington' },

family: { S: 'green' },

description:

{ S: 'Dexler is a protector of the earth and forests. He is as green as the earth and burrows into the ground for protection and extra defense.' },

protection: { N: '2' },

location\_country: { S: 'usa' },

location\_state: { S: 'kentucky' },

dragon\_name: { S: 'Dexler' } } ]

Now try triggering the scan function:

node steve\_code.js All

You should see something like this:

<... many more dragons ^ >

dragon\_name: { S: 'Atlas' } },

{ family: { S: 'green' },

damage: { N: '3' },

description:

{ S: 'Jerichombur is a dragon of mischief. His earth crushing roar can be heard for miles.' },

protection: { N: '5' },

dragon\_name: { S: 'Jerichombur' } } ]

Ok now it is all working, copy this code to your clipboard and paste it into the lambda function dragon search, overwriting it with your new tested code.

Don't forget to Save it as the new code (in the Lambda console), and use the test the DragonScan test case to trigger the scan.

You should see all the dragons returned.

Congrats, you fixed his code nad published it.

You decide to buy him that book on Test driven development from Amazon to tease him some more, before moving on to step 7

## Step 7: Learn how to do pagination in DynamoDB and to instrument the code with X-Ray - [optional info]

Steve told you that he implemented pagination in the code. He also instrumented the code with AWS X-Ray. It would be the perfect occasion to learn how that was done by looking at the code.

### 1. Pagination [optional steps, info only]

1. Look at the **scanTable** function to see how pagination was implemented.
   1. An empty array called items is created.
   2. When calling the scan function of DynamoDB, the function scanUntilDone is called.
   3. Inside that function, if the property LastEvaluatedKey exists, the attribute ExclusiveStartKey of the params object is set to that last evaluated key.
   4. The items returned by this scan are added to the items array.
   5. A new scan is called with the modified params object and with the same scanUntilDone function as the return statement thus creating a recursive loop until the property LastEvaluatedKey doesn't exists.
   6. If the property LastEvaluatedKey doesn't exists, the items of the last scan are added to the items array and that array is returned.

### 2. X-Ray [optional steps, info only]

The X-Ray SDK isn't available inside the environment provided by Lambda, so it needs to be included. In the left side of the code, you will see a folder structure with a folder named node\_modules. In that folder, you will find the aws-xray-sdk-core and its dependencies.

For instrumenting the code, the X-Ray SDK must be included:

var AWSXRay = require("aws-xray-sdk-core");

var AWS = AWSXRay.captureAWS(require("aws-sdk"));

Each time the AWS SDK is used to call a service, there will be a subsegment created about that call. There is really nothing else to do.

## Step 8: Code Optimization

Now that you fixed Steve's code, you need to do some optimizations. One of them is in the **justThisDragon** function of the **DragonSearch** Lambda function.

Whenever you run:

node steve\_code.js test Dexler

it's calling this function behind the scenes:

function justThisDragon(dragon\_name\_str, cb) {

var

params = {

ExpressionAttributeValues: {

":dragon\_name": {

S: dragon\_name\_str

}

},

FilterExpression: "dragon\_name = :dragon\_name",

TableName: "dragon\_stats"

};

DDB.scan(params, function (err, data) {

if (err) {

cb(err);

} else if (data.Items) {

cb(null, data.Items);

} else {

cb(null, []);

}

});

}

You can understand why a scan is used in the scanTable function as that must return every dragon. However, a scan is also used in the justThisDragon which only needs to return one dragon based on its name passed as the dragon\_name\_str parameter.

You are already using a filter expression which means the same amount of data is received as a payload. You have been wondering if what was said in the class about Scan versus Query is true. So you set to prove that to yourself by looking at the latency difference between the two.

You could run the code twice and see how long the Lambda function executed, but that may not give you the exact results. You could add a timer in your code to calculate it. Or better yet, you could use what you have just learned with X-Ray.

Ready for new challenges, you set yourself on the path to use X-Ray. As you learned, it's already instrumented in the code, so all you need to do is use the SDK.

Thee challenge here is that when you test the code in the Cloud IDE we run the test bypassing Xray, and when your code is in production (ie running as a lambda function) it is not by-passed and fully enabled.

The first step is to make code changes to the code in CLou d9, and test that is still works, then we will overwrite our lambda code using the console and test in the console to trigger X-Ray instrumentation.

Then when we look at the X-Ray console we should see a difference in performance.

You will first execute the code as is and record the latency seen in X-Ray. You will then modify the code of the **justThisDragon** function to use a Query. Finally, you will look at the latency for that call and compare it with what you recorded with the Scan.

Once the code is working with a Query, the next step is to look at the data. The only attributes required by the web applications from the dragon\_stats table are dragon\_name, family, protection, damage, description. However, it currently returns all of the attributes from the items. You will need to optimize the Query even further to only get those 5 attributes.

#### Step 8.0) Adjust the justThisDragon function to use a query and not a scan

1. Go to your Cloud 9 IDE, collapse any folders and close down any tabs that you are not using, and open scan\_dragon\_optimized\_query.js
2. As usual change the s thus wrting code that wil use a query instead of a scan to return just one dragon.
3. Check your code against the soution file (solution/scan\_dragon\_optimized\_query.js)
4. Ensure that you are in the right path:

cd /home/ec2-user/environment/lab4

1. Then use:

node scan\_dragon\_optimized\_query.js

You should still see this (using your new query):

Local test for a dragon called Dexler

null [ { location\_neighborhood: { S: 'bellcastle rd' },

damage: { N: '4' },

location\_city: { S: 'lexington' },

family: { S: 'green' },

description:

{ S: 'Dexler is a protector of the earth and forests. He is as green as the earth and burrows into the ground for protection and extra defense.' },

protection: { N: '2' },

location\_country: { S: 'usa' },

location\_state: { S: 'kentucky' },

dragon\_name: { S: 'Dexler' } } ]

Now we have new code we need to compare in X-Ray the difference between the old code and the new code.

Remember we have the old code in the lambda console right now, so let's test that first to instrument the old code, then after this, we will copy and paste our new code and test that, and then we compare the two in X-Ray..

Before we used API Gateway to leverage X-Ray, but this situation we are using lambda console and actually by-passing API Gateway to test it. So we must change a setting in our Lambda function that allows to instrumenting with X-Ray direvtly in Lambda.

Let's do that now:

#### Step 8.1): Check the latency of the old code (Scan and Filter)

1. Click the **Services** menu and choose **Lambda**.
2. Click on the Lambda function **DragonSearch**.
3. Scroll down to the **Debugging and error handling** card.
4. Click the check mark under **Enable active tracing**.
5. Click the **Save** button.
6. Click on the dropdown next to the Test button and select **JustOneDragon**.
7. Click the **Test** button.
8. Click on the **Monitoring** tab.
9. Click on the **View traces in X-Ray** button.
10. In the **Trace list** table, you should see a trace with an **Age** that's matching to your last test. It should be a few seconds old. If you don't see it, click the refresh button icon at the top of the page. Once you found the trace, click on its **ID**.
11. On the **DynamoDB** line, you will find the **Duration** for the Scan call. Take that number in note to use later.

#### Step 8.2): Change to a Query

It's time to do some code to potentially optimize it.

1. Click the **Services** menu and choose **Lambda**.
2. Click on the Lambda function **DragonSearch**.
3. In the **Function code** section, replace it all with your new scan\_dragon\_optimized\_query.js code.
4. You can directly modify the code in the Lambda console.
   * If you have issues, you can find the solution of the code in the file scan\_dragon-optimized-query.js located in the lab4/solution folder of your Cloud9 environment. Replace the entire code in the Lambda function with the code in that file.
5. To test the code, click the **Save** button, select **JustOneDragon** from the dropdown next to the Test button and click the **Test** button. If your code works, you should get a dragon with the dragon\_name attribute set to **Cassidiuma** returned similar to the following:
6. [
7. {
8. "location\_neighborhood": {
9. "S": "poplar st"
10. },
11. "damage": {
12. "N": "7"
13. },
14. "location\_city": {
15. "S": "colby"
16. },
17. "family": {
18. "S": "green"
19. },
20. "description": {
21. "S": "Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen's most loved and feared warrior."
22. },
23. "protection": {
24. "N": "10"
25. },
26. "location\_country": {
27. "S": "usa"
28. },
29. "location\_state": {
30. "S": "kansas"
31. },
32. "dragon\_name": {
33. "S": "Cassidiuma"
34. }
35. }
36. ]

#### Step 8.3): Latency of a Query

1. Click on the **Monitoring** tab.
2. Click on the **View traces in X-Ray** button.
3. In the **Trace list** table, you should see a trace with an **Age** that's matching to your last test. It should be a few seconds old. If you don't see it, click the refresh button icon at the top of the page. Once you found the trace, click on its **ID**.
4. On the **DynamoDB** line, you will find the **Duration** for the Query call.

Compare that number with the previous number. You may find that the timings are very similar and that would be normal for a table with a small amount of data. As your table grows, the difference will be substantial. For the moment, it may not be much or may even be higher.

Lab info: When we ran through this lab, we were noticing it going from 600ms + to around 400ms with the improved query.

#### Step 8.4): Optimize the returned data

Almost done, but It's time to do some more code optimization, namely relating to the amount of return data.

Currenty we get a lot of info back on this request but our website is only using a few attributes , so why not return less attributes in the query (the projection)?

1. Click the **Services** menu and choose **Lambda**.
2. Click on the Lambda function **DragonSearch**.
3. Open the [node AWS SDK for the query operation](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#query-property) and search for how you could specify a string that identifies one or more attributes to retrieve from the table. Maybe you have looked at this in this exercise and could take advantage of the same code if a scan and a query works similarly. Once you found it, remember that not to make the same mistake as Steve: family is a reserved keyword.
4. Rather than directly modify the code in the Lambda console. Use the Cloud 9 IDE.
5. Open scan\_dragon\_optimized\_projection.js and replace the s in order to project only the attributes you need for the website:
   1. dragon\_name.
   2. family
   3. protection
   4. damage
   5. description

Once you have improved the code, save the file.

Check you code against the solution (solution/scan\_dragon-optimized-projection.js)

Run this command

node scan\_dragon\_optimized\_projection.js test Dexler

You should see the following:

Local test for a dragon called Dexler

null [ { family: { S: 'green' },

damage: { N: '4' },

description:

{ S: 'Dexler is a protector of the earth and forests. He is as green as the earth and burrows into the ground for protection and extra defense.' },

protection: { N: '2' },

dragon\_name: { S: 'Dexler' } } ]

1. Now you have working code, copy it (scan\_dragon\_optimized\_projection.js) to your clipboard and overwrite the function code in the lambda function called DragonSearch.
2. Press save in the lambda console
3. To test the code, click the **Save** button, select **JustOneDragon** from the dropdown next to the Test button and click the **Test** button. If your code works, you should get a dragon with the dragon\_name attribute set to **Cassidiuma** and only the family, damage, protection and description attributes returned similar to the following:
4. [
5. {
6. "family": {
7. "S": "green"
8. },
9. "damage": {
10. "N": "7"
11. },
12. "description": {
13. "S": "Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen's most loved and feared warrior."
14. },
15. "protection": {
16. "N": "10"
17. },
18. "dragon\_name": {
19. "S": "Cassidiuma"
20. }
21. }
22. ]

* Click monitoring
* View traces in Xray, and click on your latest trace.

Under the DynamoDB section you should see DragonSearch has a reduced time.

Lab info: When we ran through this lab, we were noticed it was in the 170ms ballpack using this improved query and impiorved proojection.

Awseome, you have now shown Mary that you can instrument code to guide you with trouble shooting and optimize how you interact with DynamoDB.

**Congratulations!** You have completed exercise 4.

## Exercise 5

[version 1.0.4]

# Exercise: Building a Members Database using Amazon DynamoDB using the AWS Software Development Kit (AWS SDK)

In this exercise, you will learn how to develop with Amazon DynamoDB by using the AWS Software Development Kit (AWS SDK). Following the scenario provided, you will create a members area, using a DynamoDB table and using the AWS SDK. This exercise gives you hands-on experience with both Amazon DynamoDB and AWS Cloud9.

## Objectives

After completing this exercise, you will be able to use the AWS SDKs to do the following:

* Create a table and index from the Cloud 9 Terminal.
* Add items to a user database.
* Import user data from JSON (bulkWriteItem)
* Work with sensitive user data (hashing password fields) in DynamoDB.
* Create a sessions table and work with Time To Live Attributes in DynamoDB.
* Write code to validate users against DB and apply basic login/logout functionality.

## Story continued

Mary is happy that you have a working website and now wants her staff to be the only ones to see the data.

She has asked you to protect the website by providing you with a list of employees who will be the only ones that can view the website card data.

She already had a login system for her employees on premises, so for simplicity she has provided you with a JSON file with all the user data.

You remember reading about different ways to get data into DynamoDB, and you figure that (like before) using the SDK and the batchWriteItem method would be the easiest option.

## Prepare the exercise

Before you can start this exercise, you need to import some files and install some modules in the AWS Cloud9 environment that has been prepared for you.

1. Ensure you are in **Cloud9**, collapse any folders and close and files you are no longer using, ensuring that you are in the right path.

cd /home/ec2-user/environment

1. You will need get the files that will be used for this exercise, go to the Cloud 9 **bash terminal** (at the bottom of the page) and run the following wget command:

wget https://s3.amazonaws.com/awsu-hosting/edx\_dynamo/c9/dynamo-members/lab5.zip -P /home/ec2-user/environment

1. You should also see that a root folder called **dynamolab** with a lab5.zip file has been downloaded and added to your AWS Cloud9 filesystem (on the top left).
2. To unzip the workdir.zip file, run the following command:

unzip lab5.zip

This may take a few moments. In your Cloud9 filesystem.

1. To keep things clean, run the following commands to remove the zip file:

rm lab5.zip && cd lab5

1. Select the black arrow next to the lab5 folder (top left) to expand it. Notice inside this lab5 folder there is a solution folder. **Try not to peek at the solution unless you really get stuck. Always TRY to code first.**

## Step 1: Create a new users table with an index

We are goimng to need a database table that can hold information on our users (Mary's employees). This table will need to contain each employee's username, email, first name, and password.

1. Open the SDK docs and find the method for creating new tables and indexes. Find out the correct method names and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (6.17.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#createTable-property> |

You want to be able to let a user log in with their email\_address so you decide to create a Global Secondary Index (GSI), using email\_address as the Primary Key.

###### User table

| **user\_name** | **first\_name** | **email\_address (PK)** | **password** |
| --- | --- | --- | --- |
| dave937434 | Dave | [dave3423@dragonedx.com](mailto:dave3423@dragonedx.com) | sjdhjsgdjsgdjhsgd34erdtfgd |

###### Email Index (no need to project the first name)

| **user\_name** | **email\_address(PK)** | **password** |
| --- | --- | --- |
| dave937434 | [dave3423@dragonedx.com](mailto:dave3423@dragonedx.com) | sjdhjsgdjsgdjhsgd34erdtfgd |

You know how to create a table as you have done this before, however this time you will need to add an index.

**Time to write some code that creates a table and index for the users.**

1. Open up the create\_user\_table\_and\_index.js file inside the lab5 folder by double clicking on it.
2. Have the SDK docs open (as above) to help you
   1. Replace the <FMI> sections of the code in that file, so that the code creates a user table and index. You table should be called usersand and index called email\_index Use **us-east-1**.
3. Save the file
4. Go to the terminal and run your file using the respective run command below

node create\_user\_table\_and\_index.js

**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

## Confirm that your code worked.

You should see something like this in the console.

null { TableDescription:

  { AttributeDefinitions: [ [Object], [Object] ],

    TableName: 'users',

    KeySchema: [ [Object] ],

    TableStatus: 'CREATING',

    CreationDateTime: 2019-05-21T17:54:24.187Z,

    ProvisionedThroughput:

    { NumberOfDecreasesToday: 0,

       ReadCapacityUnits: 0,

       WriteCapacityUnits: 0 },

    TableSizeBytes: 0,

    ItemCount: 0,

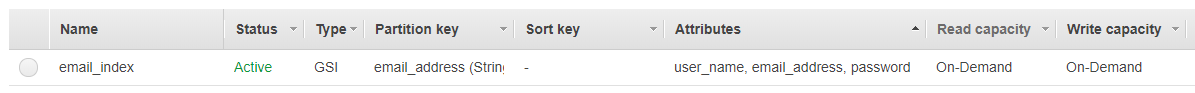
    TableArn: 'arn:aws:dynamodb:us-east-1:000000000000:table/users',

    TableId: '9c538221-b0a1-40bd-8a95-95dc4e76ed92',

    BillingModeSummary: { BillingMode: 'PAY\_PER\_REQUEST' },

    GlobalSecondaryIndexes: [ [Object] ] } }

Also visit the DynamoDB console and you will see your table (and index).



Lab tip: Even if the users table is still being ***created*** and not in an active state. you can still move on to step 2. By the time you get to step 4 where we actually upload data to it, it should be active.

## Step 2: Create a session table

You need to allow your users to log in an out, and so you will need to create a sessions table to maintain your users logged in status.

Later in this lab > exercise (3), you will enable the Time To Live feature of DynamoDB in order to expire old sessions after a few minutes. For now however, just focus on creating the table with a Primary Key and a Sort Key.

You should add a sort key on user\_name to help us (in our code) enforce lookups to only work for users that provide their user\_name\_str AND session\_id\_str.

This is what the table schema will look like (for now), where the session\_id\_str will be a randomly generated string.

###### Sessions

| **session\_id\_str (PK)** | **user\_name (SK)** | **expiration\_time (TTL)** |
| --- | --- | --- |
| dgsfdghd576s7d6yiusjghds | Dave | 1461938400 |

1. Open the SDK docs and find the method for creating new tables and indexes. Find out the correct method names and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#createTable-property> |

**Time to write some code that creates a sessions table.**

1. Open up the create\_sessions\_table.js file inside the lab5 folder by double clicking on it.
2. Close down any other tabs you are not using.
3. Have the SDK docs open (as above) to help you
   1. Replace the <FMI> sections of the code in that file, so that the code creates a sessions table with a Primary Key of session\_id and a sort key on user\_name. You table should be called sessions use **us-east-1**.
4. Save the file
5. Go to the terminal and run your file using the respective run command below

node create\_sessions\_table.js

**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

## Confirm that your code worked.

You should see something like this in the console.

null { TableDescription:

  { AttributeDefinitions: [ [Object], [Object] ],

    TableName: 'sessions',

    KeySchema: [ [Object], [Object] ],

    TableStatus: 'CREATING',

    CreationDateTime: 2019-05-21T18:05:58.386Z,

    ProvisionedThroughput:

    { NumberOfDecreasesToday: 0,

       ReadCapacityUnits: 0,

       WriteCapacityUnits: 0 },

    TableSizeBytes: 0,

    ItemCount: 0,

    TableArn: 'arn:aws:dynamodb:us-east-1:000000000000:table/sessions',

    TableId: '8f09652f-f36a-4bba-9295-1b50f5021e3f',

    BillingModeSummary: { BillingMode: 'PAY\_PER\_REQUEST' } } }

Also visit the DynamoDB console (refresh the page) and you will see your table and sort key.

1558618119307

## Step 3: Enable TTL for the sessions table

We are going to have our website front end send a session\_id\_str (token) along with their user\_name (and the rest of the payload) with every request. We can then check this sessions table to ensure that they are still logged in.

Using the **Expires At** feature we can keep this table clean and lean, not having to worry about cleaning up of sessions. They get removed automatically at any point in time that DynamoDB so chooses after that expires at time is passed.

This is your idea for an improved (expirable) sessions table schema:

###### Sessions

| **session\_id\_str (PK)** | **user\_name** | **expiration\_time (TTL)** |
| --- | --- | --- |
| gsfdghd576s7d6yiusjghds | Dave | 1461938400 |

REAL WORLD TIP: Note as there can often be a MASSIVE delay in purging expired items (up to 48 hours) using this method, you would have to check the expiration\_time at the application just in case DynamoDB didn't clean up the session in time (which it won't).

To save us writing a logout feature that deletes an item in this table, we are going to miss that out, as If they log out we can just remove the session id at the client side using JavaScript, and then they will not be able to log in anymore and all requests will fail and send them back to the login page. Letting the **Expires At** feature simply purge the item and clean out our table is a nice touch.

You can set how long this TTL is in the application code, latee on we will set it for a reasonable 20 minutes.

1. Open the SDK docs and find the method for enabling **Expires At** or **Time To Live (TTL)**. Find out the correct method names and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#updateTimeToLive-property> |

**Time to write some code that creates a table and index for the users.**

1. Open up the enable\_ttl.js file inside the lab5 folder by double clicking on it.
2. Have the SDK docs open (as above) to help you
   1. Replace the <FMI> sections of the code in that file, so that enables TTL.
3. Save the file
4. Go to the terminal and run your file using the respective run command below
5. node enable\_ttl.js

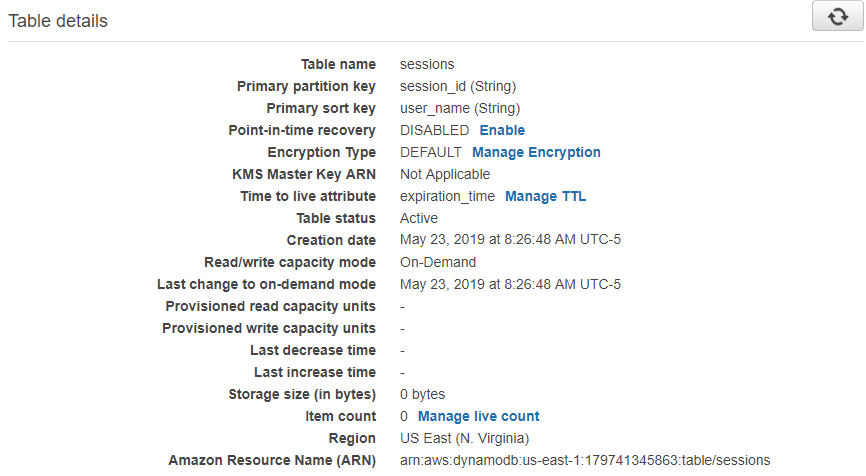
**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

## Confirm that your code worked.

You should see something like this in the console.

null { TimeToLiveSpecification: { Enabled: true, AttributeName: 'expiration\_time' } }

If you look in the DynamoDB console you will see something like this for your new sessions table:



## Step 4: Upload user data from a JSON file.

You have done something similar to this before, this should be easy, right?

MAry has given you a resources/user.json file with all the data you need for adding her employees to the database.

However the only caveat here is that Mary has provided you with temporary plain text passwords, as the plan is for her to change these out later, when she adds the **recover credentials / reset password** feature (that thankfully she has asked someone else to build later on).

The only difference verses the last time you uploaded batch items, is that you will need to hash the temporary passwords, as we should not be storing passwords in clear text in a live database.

You will notice a line in the code that hashes these temporary plain text passwords. You are going to use a library called **bcyrpt**, and as such we need to import it just like we imported the AWS-SDK. This process of importing modules to use in our code can be accomplished using Node Package Manager (NPM).

Let's do that now:

Ensure you are in the lab5 folder in the terminal

cd /home/ec2-user/environment/lab5

Then run:

npm install bcrypt

(ignore all. warnings)

Also have a look in /lab5/resources/users.json you will see that Mary has provided a list of users (employees) in JSON format:

[{

"user\_name\_str": "davey65",

"first\_name\_str": "dave",

"email\_address\_str": "dave@dragoncardgame001.com",

"temp\_password\_str": "apple"

}...]

###### Time to write some code that adds these users the the user table that you creared earlier.

1. Open up theupload\_and\_hash\_passwords.js file inside the lab5 folder by double clicking on it.
2. Have the SDK docs open (as above) to help you
   1. Replace the <FMI> sections of the code in that file, so that uploads items (batch) and hashes each password.
3. Save the file
4. Go to the terminal and run your file using the respective run command below
5. node upload\_and\_hash\_passwords.js

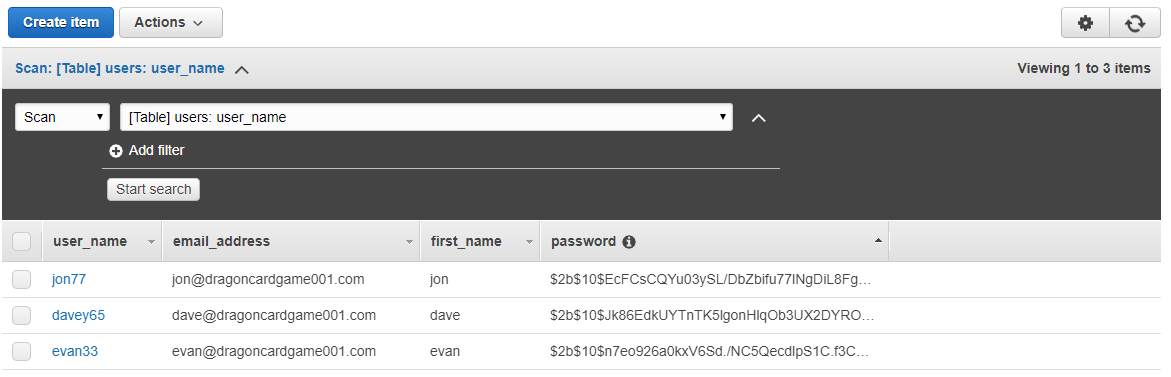
**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

## Confirm that your code worked.

You should see something like this:

[ { UnprocessedItems: {} } ]

HowFastWasThat: 309.633ms



## Step 5: Building a session protected resource

Now it's time to write code that will only allow our AJAX calls that are coming from the website to get information from DynamoDB **only** when the request comes with a **session token** that can be confirmed against our session table.

To do this we need to do a few things.

1. Firstly we need a way to log the users in.
2. Secondly we need a way to validate a provided session token before querying DynamoDB for dragon data.
3. We also need to update our website index3.html (luckily this has been done for you to save on exercise time)

We can do this by creating an API **resource** in your existing API called /login

Where the user (via the website) can pass a payload a bit like this:

{

email\_address\_str: "dave@dragoncardgame001.com",

password\_attempt\_str: "apple"

}

This new /login resource will call a new Lambda function called **login** that will hash the attempted password using **bcrypt** and compare it to the **password** field we have in the user table.

If it matches, we create a new entry in the sessions table that will expire (TTL) in 20 minutes, and return the **session token** to the website user to store in the browser. If there is no match, we proceed no further and return a "not allowed" response to the website.

By allowing our users to log in (and get logged out again after 20 minutes) from the website, future requests can be validated against this sessions table, thus protecting our data from those that are not logged in.

#### Step 5A): Create a new login function (AWS Lambda) that will log in a user.

##### Time to write some code that works with not only the user table, but the session table.

1. Open up the login.js file inside the lab5 folder by double clicking on it.
2. Ensure you are in the lab5 folder in the terminal (if you are not already).
3. cd /home/ec2-user/environment/lab5

Then run this intallation command to bring in our random string generator stuff (for our session\_id\_str's).

npm install uuid

1. Have the SDK docs open to help you
   1. Replace the <FMI> sections of the code in that file.
2. Save the file
3. Go to the terminal and run your script.

First with the wrong password:

node login.js test dave@dragoncardgame001.com coffee

You will see this fail as follows:

Local test to log in a user with email of dave@dragoncardgame001.com

$2b$10$VwKlYGzuV8tWTppFYbLihOuEJSvX1bHQebsYw4yan8lPL386t7taa coffee

password does not match email null

Now try with the correct password

node login.js test dave@dragoncardgame001.com apple

You will see something like this:

Local test to log in a user with email of dave@dragoncardgame001.com

$2b$10$VwKlYGzuV8tWTppFYbLihOuEJSvX1bHQebsYw4yan8lPL386t7taa apple

Password is correct

{ ConsumedCapacity: { TableName: 'sessions', CapacityUnits: 1 } }

AWAITED c2eef372-d727-403b-b7a6-00be76c59cd9

null { user\_name\_str: 'davey65',

session\_id\_str: 'c2eef372-d727-403b-b7a6-00be76c59cd9' }

What is interesting here, is that you now have a valid session.

If you visit the DynamoDB console and look at the session table's items (you may need to refresh), you will see the valid session in there as the only item.

You may think that as you set the session to be 1 minute in your code, you would be able to refresh the DynamoDB console again and see if it has expired after 60 seconds. However this is not how expiration in DynamoDB works. Try and think of it as a flag that is set that says "at some point anytime from now I can be safely removed from the table"

Lab Info: If you left it overnight you would see the item purged, however what you would normally do (in production) is to ensure that when we check for a valid session we are comparing the *expiration\_time* to the current time, and not just checking for existence of a session in the table (that may be flagged for removal but not yet purged).

* Now that is working let's switch the session to a workabel 20 minutes instead of 60 seconds. Go back to the code (login.js) and swap the **SESSION\_TIMEOUT\_IN\_MINUTES\_INT = 1**; on line 19 from 1 minute to 20 minutes as below.

SESSION\_TIMEOUT\_IN\_MINUTES\_INT = 20;

* Save the file.

#### We now need to create a lambda function that is based on this code.

You have already created a Lambda function before, so the concepts should be pretty be familiar to you, the only difference however is this time, you will use the SDK.

You will need to package up your recently adjusted and tested code along with the **UUID** and **Bcrypt** modules, and build a new Lambda function in N.Virginia called **LoginEdXDragonGame**.

There are couple of tasks we need to take care of first, such as creating a **role** that Lambda can use to talk to DynamoDB and allowing it to talk to CloudWatch logs and Xray.

We are going to use a new role just for login called **login-for-dragons-role**.

#### Step 5B): Create a new IAM role for use with Lambda

By creating a role for the Lambda function you are about to create. You are essentially allowing your function code to read and write to DynamoDB and write logs to CloudWatch Logs and Xray.

1. From your Cloud9 dashboard choose **AWS Cloud9** in the upper left.
2. Then choose **Go to your dashboard**.
3. Go to **services** and choose **IAM**.
4. Choose **Roles** and choose **create role**.
5. Select **Lambda** and choose **Next: Permissions**.
6. In the **search** box type in **Dynamo** and select the **checkbox** next to AmazonDynamoDBFullAccess.
7. Again in the **search** box type in **Lambda** and select the **checkbox** next to AWSLambdaBasicExecutionRole.
8. And again in the **search** box type in **AWSXrayWriteOnlyAccess** and select the **checkbox** next to AWSXrayWriteOnlyAccess.
9. Choose **Next: Tags** and leave it as is. Select **Next:Review**.
10. Type the name login-for-dragons-role in the **Role name** box. Then choose **Create role**.
11. Click on the login-for-dragons-role hyperlink at the top of the page.
12. Copy the Role ARN, something like this: (as you will need this as a <FMI> in the next code section)
13. arn:aws:iam::xxxxxx:role/login-for-dragons-role

#### Step 1C): Create the Lambda package with all its dependencies.

1. Zip a package using the Cloud 9 terminal.
2. cd /home/ec2-user/environment/lab5
3. Install the packages into a folder called node\_modules
4. npm install --prefix /home/ec2-user/environment/lab5 uuid bcrypt
5. Ignore any warnings.
6. Now zip it all up
7. zip -r login.zip node\_modules login.js

You will see that a **login.zip** file has been create in the following path /home/ec2-user/environment/lab5/ in Cloud9.

**Congrats**, you have your lambda function all packaged up

Now we need to publish it.

## Step 6: Logins and users - The publish step

1. Open the SDK docs and find the method for creating new lambda functions. Find out the correct method names and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Lambda.html#createFunction-property> |

**Time to write some code that creates a sessions table.**

1. Open up the create\_and\_publish\_login.js file inside the lab5 folder by double clicking on it.
2. Close down any other tabs, and collapse any folders that you are not using.
3. Have the SDK docs open (as above) to help you
   1. Replace the <FMI> sections of the code in that file, so that the code creates a new function from the ZIP file. Use **us-east-1**.
4. Save the file
5. Go to the terminal and run your file using the run command below. (you should be in the /lab5 path in the Cloud9 terminal).
6. node create\_and\_publish\_login.js

## Confirm that your code worked.

You should see something like this:

{ FunctionName: 'LoginEdXDragonGame',

FunctionArn: 'arn:aws:lambda:us-east-1:<000000000000>:function:LoginEdXDragonGame',

Runtime: 'nodejs8.10',

Role: 'arn:aws:iam::<000000000000>:role/login-for-dragons-role',

Handler: 'create\_and\_publish\_login.handler',

CodeSize: 1034766,

Description: 'Login functionality',

Timeout: 30,

MemorySize: 128,

LastModified: '2019-05-23T19:17:11.761+0000',

CodeSha256: 'Afgtyddw6mwaiPrgIOvyH5uTWB6X4x9lKWUpvo78FSo=',

Version: '1',

KMSKeyArn: null,

TracingConfig: { Mode: 'PassThrough' },

MasterArn: null,

RevisionId: '18c3abf2-5td4-4372-b52a-18b1d2eddfd6' }

Also you could check the Lambda console to see the function. It will be called LoginEdXDragonGame

**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

#### Step 6A): Test your Lambda function

1. Choose **Services** and search for **Lambda**.
2. Choose the **LoginEdXDragonGame** function.
3. Choose **Test**.
4. Leave the **Event template** as **Hello World**.
5. For **Event name** use fakePassword.
6. Paste the following into event box:
7. {
8. "email\_address\_str": "dave@dragoncardgame001.com",
9. "attempted\_password\_str": "spaceship"
10. }
11. Choose **Create**.
12. Choose **Test**.
13. You should see it fail with: (expand details)
14. {
15. "errorMessage": "password does not match email"
16. }
17. Now use the correct password. Create a new test under **Configure test events** (in the test case dropdown) and call it **realPassword** and paste the following contents.
18. {
19. "email\_address\_str": "dave@dragoncardgame001.com",
20. "attempted\_password\_str": "apple"
21. }
22. Run the test and you should see something like this:
23. {
24. "user\_name\_str": "davey65",
25. "session\_id\_str": "9f15f3ef-4700-43e7-b602-f11f8b892727"
26. }

Every time you press Test it will create a new session\_id\_str

Now everything is working with respect to the login, we can adjust our API gateway to allow us to call this from the website and get a user to log in.

#### Step 6B): Add the login method to our API Gateway

1. Choose **Services** and search for **API**.
2. Choose the **DragonSearchAPI**.
3. Choose **Actions** and **Create Resource**.
4. Under **Resource Name** type in login. You will see the path is also updated /login.
5. Choose **Create Resource**.(Leave eveything unchecked)
6. With /login selected (highlighted)
7. Choose **Actions** and **Create Method**.
8. Choose **POST** and click the checkmark icon next to it.
9. Leave the **Integration type** as **Lambda Function**.
10. Leave Lambd proxy **unchecked**.
11. In the **Lambda Function** type in **LoginEdxDragonGame**.
12. Choose custom timeout of 10000 (10 seconds)
13. Choose **Save**.
14. Choose **OK** at at add permissions pop-up.
15. Choose **TEST**.

Remove everything from the **Request Body**.

Now press **Test** and you will see:

{

"errorMessage": "no credentials passed"

}

Now try with an incorrect password: Replace the currently empty Request Body with this then press test.

{

"email\_address\_str": "dave@dragoncardgame001.com",

"attempted\_password\_str": "yellow"

}

You should see the following:

{

"errorMessage": "password does not match email"

}

Now test with correct password. Paste the following into the **Request Body**:

{

"email\_address\_str": "dave@dragoncardgame001.com",

"attempted\_password\_str": "apple"

}

You should see something like the following:

{

"user\_name\_str": "davey65",

"session\_id\_str": "16e7dec8-963a-4645-b454-4f245c8b0067"

}

1. Select the /login under **Resources** (so it is again highlighted) and choose **Actions** and **Enable CORS**.
2. Check **DEFAULT 4XX** and **DEFAULT 5XX** (ignore any warnings).
3. Choose **Enable CORS and replace existing CORS headers**.
4. Choose **Yes, replace existing values**.
5. Choose **Actions** and **Deploy API**.
6. Choose **prod** and choose **Deploy**.

#### Step 6C): Test the website.

1. Go to your website but use /index3.html instead of /index2.html.
2. You should be presented with a login screen.
3. Try and login to the website with a fake password:
4. dave@dragoncardgame001.com
5. yellow

You should see a **"credentials invalid"** message. Which is what we would expect.

1. Now try using the correct password:
2. dave@dragoncardgame001.com
3. apple

Now you should be able to view dragon cards and you should also see a logout davey65 button top right.

**Congrats** you are only allowing logged in users from viewing card data on your website

## Step 7: Lockdown

The website is stopping unauthenticated users from access the content, but the API that is called behind the scenes to access card data is not protected.

Nothing is stopping the API from being called **directly** and by-passing the website login protection,

For example try this the Cloud 9 terminal using your API gateway URL instead of the <FMI>. (or view the old index2.html ;))

curl -L -XPOST "<FMI>" -d '{"dragon\_name\_str": "Nightingale"}'

You should see the following:

[{"family":{"S":"black"},"damage":{"N":"4"},"description":{"S":"Nightingale uses her song to lull and seduce her opponents. She is deadly."},"protection":{"N":"6"},"dragon\_name":{"S":"Nightingale"}}]

Not ideal right? It would't take very much for someone to hack the site to get the card information.

So we need to update our DragonSearch lambda function so that when a request comes in to it, it will reject anything without a valid (checkable) username / session token (which the browser now has, because of course the user is logged in and the website kept that information in local storage).

Locking this down is a quick fix, as there is no need to redeploy our API GW endpoint. We simply update the Lambda function code that it is point it. We could edit the file in Cloud 9 and repackage the lambda function and publish a new version,, however as this is only a few lines of code, we are going to edit the code in the lambda console directly and test it there.

1. Navigate back to the Lambda console choosing **Services** and searching for **Lambda**.
2. Choose the **DragonSearch** function.
3. We are not going to make you folks code this step
4. Just replace the contents of **index.js** with the provided code at/lab5/resources/protected\_dragon\_search.js.
5. All this code does is check for a valid session before moving on.
6. Choose **Save**.
7. Run one of the old test cases that you created earlier called justOneDragon
8. You should get: Which means it is now protected. #win
9. {
10. "errorType": "string",
11. "errorMessage": "not allowed",
12. "trace": []
13. }

Finally test in the command line that your new API is using this Lambda function. This time the command should fail: (use your API gateway URL instead of the )

curl -L -XPOST "<FMI>" -d '{"dragon\_name\_str": "Nightingale"}'

With a message saying "nope", not allowed in.

{"errorType":"string","errorMessage":"not allowed","trace":[]}

Also index2.html will no longer display dragons, which is what we would expect as no credentials are being passed in that version of the website.

**Congrats** you have locked down your website to members of the site only. Mary will be happy!

## Exercise 6

[version\_1.0.4]

# Exercise: Secure Amazon DynamoDB using the AWS Software Development Kit (AWS SDK)

## Overview

In this exercise, you will learn how to develop with Amazon DynamoDB by using the AWS Software Development Kit (AWS SDK). Following the scenario provided, you will learn how to apply the least privilege principle to access DynamoDB and communicate with DynamoDB without going over the Internet. This exercise gives you hands-on experience with AWS Identity and Access Management and Amazon Virtual Private Cloud Gateway Endpoints.

## Objectives

After completing this exercise, you will be able to:

* Create Fine-Grained Access Control IAM Policy Conditions for Amazon DynamoDB
* Move a Lambda function inside a VPC
* Create a VPC Gateway Endpoint for DynamoDB

## Story continued

Steve has just notified you that sensitive data will be stored in the dragon\_stats DynamoDB table. That sensitive data shouldn't be readable by the Lambda function as he's afraid that a malicious developer could change the code to read that sensitive data. He reviewed some of the IAM Policies that are applied to the IAM Role, call-dynamodb-role, used by the Lambda function that runs your code and found out that they are too permissive.

He wants you to follow the least privilege principle mandated by the Security Team and only allows:

* specific actions executed by the application
* querying for specific attributes required by the application

As you know which action types and attributes are used in your application, he thought it would make more sense to leave this task to you. So you will need to read the code to find that data and create the most Fine-Grained IAM Policy you can without breaking the application.

He also wants you to move the Lambda function inside of a Virtual Private Cloud (VPC) that doesn't permit any access to the Internet. He mentioned that the reason for doing that is because another developer is working on a feature that will only be available inside the VPC. As you know, DynamoDB is available over the Internet and that's what you have been using so far. So he asked you to find a way for the Lambda function to communicate to DynamoDB from within the VPC without going over the Internet. That's when you remembered about VPC Gateway Endpoints. It's a good time to implement this while the Lambda function isn't in production.

## Prepare the exercise

Before you can start this exercise, you need to import some files and install some tools in the AWS Cloud9 environment that you have created.

1. From the AWS Management Console, go to the **Services** menu and choose **Cloud9**.
2. Choose **Open IDE** to open the AWS Cloud9 environment. Close down any tabs you are not using and collapse any inactive folders.
3. Ensure you are in the base path in your Cloud9 terminal.
4. cd /home/ec2-user/environment
5. Install the **jq** package, run the following command

sudo yum install jq

# if it says is this ok, select y (for yes)

1. To get the files that will be used for this exercise, go to the Cloud 9 **bash terminal** and run the following wget command:

wget https://s3.amazonaws.com/awsu-hosting/edx\_dynamo/c9/dynamo-secure/lab6.zip -P /home/ec2-user/environment

You should also see that a root folder called **dynamolab** with a lab6.zip file has been downloaded and added to your AWS Cloud9 filesystem (on the top left).

1. To unzip the lab6.zip file, run the following command:

unzip lab6.zip

In your Cloud9 filesystem, you should see a lab6 folder.

1. To keep things clean, run the following commands to remove the zip:

rm lab6.zip

cd lab6

echo "done"

## Step 1: Create a new IAM Policy and Role

The current IAM Role associated to the **DragonSearch** Lambda function may end up being used by more than one function. To ensure that the other functions continue to work, you will create a specific IAM Role just for this Lambda function.

You will start with creating your own IAM Policy using generic statements using the Visual Policy Editor and look at the JSON code.

You will then create the new Role and associate your IAM Policy and the **AWSLambdaVPCAccessExecutionRole** AWS managed Policy. The AWSLambdaVPCAccessExecutionRole Policy will be used later in this exercise to allow Lambda to run inside your VPC.

Finally, you will modify the **DragonSearch** Lambda function to use your new Role and test it before moving to the next step.

### Step 1A): Create new Policy

1. From **Cloud9**, click the **AWS Cloud9** button next to File.
2. Select **Go To Your Dashboard**.
3. Click on the **Services** menu and choose **IAM**.
4. Click on the **Policies** link from the left side menu.
5. Click on the **Create policy** button.
6. Click on the **Choose a service** link to expand it.
7. In the **Find a service** bar, input DynamoDB.
8. Click on the **DynamoDB** link to select it.
9. In the **Actions** section, click the checkbox next to **All DynamoDB actions**.
10. Click on the **Resources** section to expand it.
11. Select the radio button next to **All resources**.
12. Click on the **Review policy** button.
13. In the **Name** field, input edx-ddb-dragonsearch-policy.
14. Click on the **Create policy** button.

### Step 1B): Create new Role

1. Click on the **Roles** link from the left side menu.
2. Click on the **Create role** button.
3. Click on the **Lambda** link under the **Choose the service that will use this role** section.
4. Click on the **Next: Permissions** button.
5. In the **Search** bar, input edx-ddb-dragonsearch-policy.
6. Click the checkbox next to **edx-ddb-dragonsearch-policy**.
7. In the **Search** bar, input AWSLambdaVPCAccessExecutionRole.
8. Click the checkbox next to **AWSLambdaVPCAccessExecutionRole**.
9. Click on the **Next: Tags** button.
10. Click on the **Next: Review** button.
11. In the **Role name** field, input edx-ddb-dragonsearch-role.
12. Click on the **Create role** button.

### Step 1C): Associate the new Role to Lambda and Test

1. Click on the **Services** menu and choose **Lambda**.
2. Click on the **DragonSearch** lambda function link.
3. Scroll down to the **Execution role** card.
4. Click on the **Existing role** dropdown.
5. In the search bar, input edx-ddb-dragonsearch-role.
6. Click on the **edx-ddb-dragonsearch-role** entry.
7. Click on the **Save** button (top of page)
8. In the dropdown next to the Test button, select **JustOneDragon**.
9. Click on the **Test** button.

The details output should be the following:

{

"errorType": "string",

"errorMessage": "not allowed",

"trace": []

}

This is actually what you should expect, as we locked down access to this API to only allow access to logged in users.

To really be sure that this new policy still works, we should create a test case that sends a session.

First get a new valid session: Using the Cloud 9 terminal.

You may need to have the lab 5 folder still in your Cloud9 IDE for this to work

node ../lab5/solution/login.js test dave@dragoncardgame001.com apple

This should give you a new session\_id\_str andexpose the user\_name\_str

$2b$10$Q5.0VU2CA5JFnc0r6hSXfeNFpm2XoVXlKMVnniR7pPivIMb7wvoVy apple

Password is correct

{ ConsumedCapacity: { TableName: 'sessions', CapacityUnits: 1 } }

AWAITED 18b3dcf4-06b1-4d2e-98ad-a70f4e0b40ac

null { user\_name\_str: 'davey65',

 session\_id\_str: '18b3dcf4-06b1-4d2e-98ad-a70f4e0b40ac' }

Now, in the lambda console for **DragonSearch**

Press **configure test event** in the (test dropdown).

Create a new test event called withSession

Replace the content with this: Using the session\_id\_str you just got.

{

"user\_name\_str": "davey65",

"session\_id\_str": "<FMI>",

"dragon\_name\_str": "Dexler"

}

Click test and you shoud see:

[

{

"family": {

"S": "green"

},

"damage": {

"N": "4"

},

"description": {

"S": "Dexler is a protector of the earth and forests. He is as green as the earth and burrows into the ground for protection and extra defense."

},

"protection": {

"N": "2"

},

"dragon\_name": {

"S": "Dexler"

}

}

]

**Excellent**!

You have now created a new Policy, a new Role and associated it to your Lambda function. By testing it and verifying the output of the test, you can now conclude that your new Role and Policy works. It's time to apply the least privilege principle to you new Policy.

## Step 2: Modify the IAM Policy to apply the Least Privilege Principle

Amazon DynamoDB doesn't work the same way as typical databases like the ones under the Amazon Relation Database Service where you are still responsible for creating the Users and Permissions inside the database. With Amazon DynamoDB, AWS IAM is still used to control the authentication and authorization thus liberating the database from this burden. This means that the IAM Policy associated with the authentication entity you are using must follow the least privilege principle. The same idea where you wouldn't use the Root/Master user for your database inside your application because its permissions are too powerful, you shouldn't grant all access to all of DynamoDB or to your Table if your application doesn't require it.

In this task, you will first review the code of the **DragonSearch** Lambda function to determine what DynamoDB API calls/actions are required. You will also look for all the attributes that are returned so you can lock down the IAM Policy even more.

You will then modify the **edx-ddb-dragonsearch-policy** IAM Policy to apply Fine-Grained Access Controls on your Policy. To do this, IAM Policy Conditions will be used. You should review the docs to ensure you understand their usage.

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/specifying-conditions.html

Finally, you will test the Lambda function to make sure the withSession test still works.

### Step 2A): Review the code of DragonSearch

1. Click on the **Services** menu and choose **Lambda**.
2. Click on the **DragonSearch** lambda function link.
3. Review the code of the **justThisDragon** and **scanTable** functions. Note all of the API calls made to the DynamoDB service. Also, take a note of all the attributes requested by those API calls. Finally, note the name of the DynamoDB table used for each of those calls.

### Step 2B): Find the ARN of the DynamoDB table

You need to get the Amazon Resource Name (ARN) of the DynamoDB table as it will be used in the Policy.

1. Click on the **Services** menu and choose **DynamoDB**.
2. Click on the **Tables** link in the left menu.
3. Click on the name of the DynamoDB table that is used in the code. dragon\_stats
4. At the bottom of the **Table details** section, you will find the **Amazon Resource Name (ARN)** field. Take a note of this ARN which should look like the following: arn:aws:dynamodb:us-east-1:xxxxxxxxx:table/dragon\_stats.

### Step 2C): Modify your IAM Policy

1. Click on the **Services** menu and choose **IAM**.
2. Click on the **Policies** link in the left menu.
3. In the **Search** bar, input edx-ddb-dragonsearch-policy.
4. Click on the **edx-ddb-dragonsearch-policy** link.
5. Click on the **Edit policy** button.

If you are familiar with IAM Policies and prefer to do this yourself, feel free to click on the **JSON** tab and edit the policy manually based on what you found in there.

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/specifying-conditions.html.

You can skip to the next task to test your Lambda function.

1. Click on the **DynamoDB** section to expand it.
2. The Service is set to DynamoDB which is what you want it to be.
3. Click on the **Actions** section to expand it.
4. Remove the checkmark next to **All DynamoDB actions**.
5. Click on the breadcrumb next to **Read** as your Lambda function is only reading from DynamoDB.
6. Place a checkmark next to all of the API calls made to DynamoDB used in the code.
7. Select the two methods that your code uses: **Query** and **Scan**.
8. Click on the **Resources** section to expand it.
9. Click the radio button next to **Specific**.
10. In the **table** section, click on the **Add ARN** link.
11. In the **Specify ARN for table** field, **replace the entire content** with the **ARN of the DynamoDB table** that you found in the previous section. It will automatically populate the entries for Region, Account and Table name.
12. Click on the **Add** button.
13. Click on the **Request conditions** section to expand it.
14. Click on the **Add condition** link.
15. In the **Condition key** dropdown, select **dynamodb:Attributes** near the bottom of the list to specify a list of attributes that the user can get back.
16. In the **Qualifier** dropdown, select **For all values in request** to match every values in the specified **ProjectionExpression** parameter in the API call to DynamoDB.
17. In the **Operator** dropdown, select **StringEquals**.
18. In the **Value** field, input the value of an attribute name that you found while looking at the code. Click on the **Add another condition value** and input the next attribute name. Repeat this step until you have filled all the attributes. The order doesn't have any importance, but make sure that all the attributes are listed. They should be "**dragon\_name**", "**family**", "**protection**", "**damage**","**description**"
19. Click on the **Add** button.
20. To enforce the use of the **ProjectionExpression** parameter in the API Call, you need to specify that the dynamodb:Select attribute is set to SPECIFIC\_ATTRIBUTES. If you don't do that and you only enforce the attributes, then if the **ProjectionExpression** parameter isn't used, the attributes list won't be enforced.
    1. Click on the **Add another condition** link.
    2. In the **Condition key** dropdown, select **dynamodb:Select** at the bottom of the list.
    3. Leave the **Qualifier** to **Default**.
    4. In the **Operator** dropdown, select **StringEquals**.
    5. In the **Value** field, input SPECIFIC\_ATTRIBUTES.
    6. Click on the **Add** button.
21. Click on the **Review policy** button.
22. Click on the **Save changes** button.
23. You policy should look similar to this:

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "VisualEditor0",

"Effect": "Allow",

"Action": [

"dynamodb:Scan",

"dynamodb:Query"

],

"Resource": "arn:aws:dynamodb:us-east-1:xxxxxxxxx:table/dragon\_stats",

"Condition": {

"ForAllValues:StringEquals": {

"dynamodb:Attributes": [

"dragon\_name",

"family",

"protection",

"damage",

"description"

]

},

"StringEqualsIfExists": {

"dynamodb:Select": "SPECIFIC\_ATTRIBUTES"

}

}

}

]

}

What this is doing is preventing a malicious coder requesting more data back than we want them to have, We don't need the website have access to the location\_\* information for example, so locking this down at the database level using IAM least privilege, prevents a malicious coder by-passing the current coded projection.

### Step 2D): Test the Lambda function (again, using the new policy)

1. You will need a valid sesison again in a sec for use in the est case so grab a new session now ready. Use the Cloud 9 terminal to get a new session.
2. node ../lab5/solution/login.js test dave@dragoncardgame001.com apple
3. This should give you a new session like before:
4. Local test to log in a user with email of dave@dragoncardgame001.com
5. $2b$10$Q5.0VU2CA5JFnc0r6hSXfeNFpm2XoVXlKMVnniR7pPivIMb7wvoVy apple
6. Password is correct
7. { ConsumedCapacity: { TableName: 'sessions', CapacityUnits: 1 } }
8. AWAITED a1080978-ab39-4666-ac0c-7d1f98529f4c
9. null { user\_name\_str: 'davey65',
10. session\_id\_str: 'a1080978-ab39-4666-ac0c-7d1f98529f4c' }
11. Click on the **Services** menu and choose **Lambda**.
12. Click on the **DragonSearch** lambda function link.
13. In the dropdown next to the Test button, edit the withSession test payload to include the new session where you see the <FMI>:
14. {
15. "user\_name\_str": "davey65",
16. "session\_id\_str": "<FMI>",
17. "dragon\_name\_str": "Fireball"
18. }
19. Click on the **Test** button.

The output should fail and look like the following:

{

"errorType": "string",

"errorMessage": "nope",

"trace": []

}

If you look at the log ourtput. Section of the. Execution result, you will see somthing like this:

START RequestId: a77751dd-d7d0-485c-b997-6219ab3f05b9 Version: $LATEST

2019-06-11T20:06:27.983Z a77751dd-d7d0-485c-b997-6219ab3f05b9 INFO To run a Local test in Cloud 9 use `node scan\_dragons.js test`

2019-06-11T20:06:27.983Z a77751dd-d7d0-485c-b997-6219ab3f05b9 INFO running in Lambda

2019-06-11T20:06:27.983Z a77751dd-d7d0-485c-b997-6219ab3f05b9 INFO davey65 18b3dcf4-06b1-4d2e-98ad-a70f4e0b40ac

2019-06-11T20:06:29.024Z a77751dd-d7d0-485c-b997-6219ab3f05b9 INFO { AccessDeniedException: User: arn:aws:sts::628920026067:assumed-role/edx-ddb-dragonsearch-role/DragonSearch is not authorized to perform: dynamodb:Query on resource: arn:aws:dynamodb:us-east-1:628920026067:table/sessions

at Request.extractError (/var/runtime/node\_modules/aws-sdk/lib/protocol/json.js:51:27)

at Request.callListeners (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:106:20)

at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:78:10)

at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/request.js:683:14)

at Request.transition (/var/runtime/node\_modules/aws-sdk/lib/request.js:22:10)

at AcceptorStateMachine.runTo (/var/runtime/node\_modules/aws-sdk/lib/state\_machine.js:14:12)

at /var/runtime/node\_modules/aws-sdk/lib/state\_machine.js:26:10

at Request.<anonymous> (/var/runtime/node\_modules/aws-sdk/lib/request.js:38:9)

at Request.<anonymous> (/var/runtime/node\_modules/aws-sdk/lib/request.js:685:12)

at Request.callListeners (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:116:18)

message:

'User: arn:aws:sts::628920026067:assumed-role/edx-ddb-dragonsearch-role/DragonSearch is not authorized to perform: dynamodb:Query on resource: arn:aws:dynamodb:us-east-1:628920026067:table/sessions',

code: 'AccessDeniedException',

time: 2019-06-11T20:06:28.904Z,

requestId: '96GLOQ57HC2VKE7K3IC25D0KV3VV4KQNSO5AEMVJF66Q9ASUAAJG',

statusCode: 400,

retryable: false,

retryDelay: 24.719242158357947 }

2019-06-11T20:06:29.024Z a77751dd-d7d0-485c-b997-6219ab3f05b9 ERROR Invoke Error {"errorType":"Error","errorMessage":"nope","stack":["Error: nope"," at \_homogeneousError (/var/runtime/CallbackContext.js:12:12)"," at postError (/var/runtime/CallbackContext.js:29:51)"," at callback (/var/runtime/CallbackContext.js:40:7)"," at /var/runtime/CallbackContext.js:103:16"," at /var/task/index.js:24:24"," at Response.<anonymous> (/var/task/index.js:57:21)"," at Request.<anonymous> (/var/runtime/node\_modules/aws-sdk/lib/request.js:364:18)"," at Request.callListeners (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:106:20)"," at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:78:10)"," at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/request.js:683:14)"]}

END RequestId: a77751dd-d7d0-485c-b997-6219ab3f05b9

REPORT RequestId: a77751dd-d7d0-485c-b997-6219ab3f05b9 Duration: 1229.91 ms Billed Duration: 1300 ms Memory Size: 128 MB Max Memory Used: 94 MB

So this is good news and bad news

The good news is that the imporved policy is working! #win

The bad news is that this is reminding us (doh!) that the session tables is contacted in this code too. We we need to modify our policy to include access to the sessions table. otherwise our IAM polciy is so tight that it won't let the code talk to the authentication resource table calledsessions, basically blocking everyone!

Let's edit our policy some more:

* Head over to **IAM**
* Click on **Policies**
* Select the edx-ddb-dragonsearch-policy policy
* Click the **JSON** tab.
* Replace the contents with the following, remember to replace your account number in your ARN where you see the three <FMI>s

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "VisualEditor0",

"Effect": "Allow",

"Action": [

"dynamodb:Scan",

"dynamodb:Query"

],

"Resource": "arn:aws:dynamodb:us-east-1:628920026067:table/dragon\_stats",

"Condition": {

"StringEquals": {

"dynamodb:Select": "SPECIFIC\_ATTRIBUTES"

},

"ForAllValues:StringEquals": {

"dynamodb:Attributes": [

"dragon\_name",

"family",

"protection",

"damage",

"description"

]

}

}

},

{

"Sid": "VisualEditor1",

"Effect": "Allow",

"Action": [

"dynamodb:Query"

],

"Resource": "arn:aws:dynamodb:us-east-1:628920026067:table/sessions"

}

]

}

Notice we added one more statement to allow a query. We could have tightened up the projection here to, to prevent the "expires at attribute being retuned", however this is not really a big deal and would involve editing your code and adding a projection.

Ok, let's try and test again using the lambda console.

* In the dropdown next to the Test button, select **JustOneDragon**.
* Click on the **Test** button.

The output should be like the following:

[

{

"family": {

"S": "red"

},

"damage": {

"N": "2"

},

"description": {

"S": "Fireball is a young dragon in training. He is learning how to control his fire, but is still lethal."

},

"protection": {

"N": "6"

},

"dragon\_name": {

"S": "Fireball"

}

}

]

Ok now it is all working, let's pretend to be the malicious coder and get the location\_city attribute back in our results.

Let's try replacing the **ProjectionExpression** of the **justThisDragon** function to include another value location\_city:

* In your Lambda console, replace a line of code (line 80) which is currently this::

ProjectionExpression: "dragon\_name, #family, protection, damage, description",

* Change it to this (to include location)

ProjectionExpression: "dragon\_name, #family, protection, damage, description, location\_city",

AND

Replace line 101 of the **scanTable** function.

From:

ProjectionExpression: "dragon\_name, #family, protection, damage, description"

To

ProjectionExpression: "dragon\_name, #family, protection, damage, description, location\_city"

NOTE: NO trailing comma on line 101.

* Click on the **Save** button.
* Click on the **Test** button while making sure that **withSession** is selected.
* TIP: Hopefully you still have a valid session for the test case *withSession*. If not grab one again like you did before and pass in a valid *session\_id\_str* into that payload then click test.

With a valid session, you should get an **Runtime.ExitError** similar to the following when you run the withSession test case.

{

"errorType": "Runtime.ExitError",

"errorMessage": "RequestId: 02e19c49-ee86-4432-ac14-5a58ef7b0db4 Error: Runtime exited with error: exit status 129"

}

If we take a closer look at the logs:

START RequestId: 417f1e42-e03b-428d-956a-7478bf82dd1b Version: $LATEST

2019-06-11T20:48:50.710Z 417f1e42-e03b-428d-956a-7478bf82dd1b INFO To run a Local test in Cloud 9 use `node scan\_dragons.js test`

2019-06-11T20:48:50.748Z 417f1e42-e03b-428d-956a-7478bf82dd1b INFO running in Lambda

2019-06-11T20:48:50.748Z 417f1e42-e03b-428d-956a-7478bf82dd1b INFO davey65 0a9d4731-cd54-4040-b516-60e50641514f

2019-06-11T20:48:51.368Z 417f1e42-e03b-428d-956a-7478bf82dd1b INFO match

2019-06-11T20:48:51.368Z 417f1e42-e03b-428d-956a-7478bf82dd1b INFO Full scan all

2019-06-11T20:48:51.591Z 417f1e42-e03b-428d-956a-7478bf82dd1b ERROR Uncaught Exception {"errorType":"AccessDeniedException","errorMessage":"User: arn:aws:sts::628920026067:assumed-role/edx-ddb-dragonsearch-role/DragonSearch is not authorized to perform: dynamodb:Scan on resource: arn:aws:dynamodb:us-east-1:628920026067:table/dragon\_stats","code":"AccessDeniedException","stack":["AccessDeniedException: User: arn:aws:sts::628920026067:assumed-role/edx-ddb-dragonsearch-role/DragonSearch is not authorized to perform: dynamodb:Scan on resource: arn:aws:dynamodb:us-east-1:628920026067:table/dragon\_stats"," at Request.extractError (/var/runtime/node\_modules/aws-sdk/lib/protocol/json.js:51:27)"," at Request.callListeners (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:106:20)"," at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:78:10)"," at Request.emit (/var/runtime/node\_modules/aws-sdk/lib/request.js:683:14)"," at Request.transition (/var/runtime/node\_modules/aws-sdk/lib/request.js:22:10)"," at AcceptorStateMachine.runTo (/var/runtime/node\_modules/aws-sdk/lib/state\_machine.js:14:12)"," at /var/runtime/node\_modules/aws-sdk/lib/state\_machine.js:26:10"," at Request.<anonymous> (/var/runtime/node\_modules/aws-sdk/lib/request.js:38:9)"," at Request.<anonymous> (/var/runtime/node\_modules/aws-sdk/lib/request.js:685:12)"," at Request.callListeners (/var/runtime/node\_modules/aws-sdk/lib/sequential\_executor.js:116:18)"],"message":"User: arn:aws:sts::628920026067:assumed-role/edx-ddb-dragonsearch-role/DragonSearch is not authorized to perform: dynamodb:Scan on resource: arn:aws:dynamodb:us-east-1:628920026067:table/dragon\_stats","time":"2019-06-11T20:48:51.591Z","requestId":"E60G2SNQ2E8PT44H6GSK56FRABVV4KQNSO5AEMVJF66Q9ASUAAJG","statusCode":400,"retryable":false,"retryDelay":26.91456449885179}

END RequestId: 417f1e42-e03b-428d-956a-7478bf82dd1b

REPORT RequestId: 417f1e42-e03b-428d-956a-7478bf82dd1b Duration: 1752.91 ms Billed Duration: 1800 ms Memory Size: 128 MB Max Memory Used: 44 MB

RequestId: 417f1e42-e03b-428d-956a-7478bf82dd1b Error: Runtime exited with error: exit status 129

Runtime.ExitError

You can see that IAM prevents us from accessing that location\_str attribute. #win

So that we don't break our website we are going to revert these lines now:

Replace line 101 of the **scanTable** function.

From:

ProjectionExpression: "dragon\_name, #family, protection, damage, description, location\_city"

To

ProjectionExpression: "dragon\_name, #family, protection, damage, description"

no trailing comma

**And**

Replace a line of code (line 80) which is currently this::

ProjectionExpression: "dragon\_name, #family, protection, damage, description, location\_city",

* Change it to this (to remove the location)

ProjectionExpression: "dragon\_name, #family, protection, damage, description, location\_city",

Do a quick test to make sure it stil works like before.

* Test withSesssion

You should get a list of all dragons.

Ok ready to move on, as you have successfully implemented the least privilege principle by using Fine-Grained Access Controls on your IAM Policy for your Lambda function.

It's time to work on the next task from Steve, protecting access to only allow CODE acting **inside** the VPC to access DynamoDB.

## Step 3: Configure VPC resources and Lambda to use it

You will first test the **DragonSearch** function using the CLI to make sure it's in a working state.

Then, you need to create a Virtual Private Cloud, a Subnet and a Security Group that will be used by the Lambda function. To create those resources, the AWS CLI will be used in your Cloud9 environment via a script called lab6/resources/create\_vpc\_resources.sh. You will need to take note of the output of that script as it will be used in the next steps. Feel free to look at the lab6/resources/create\_vpc\_resources.sh script to understand how it was done, however this isn't the purpose of the exercise.

You will then execute a CLI command to instruct Lambda to attach your Lambda function to the VPC you created.

Finally, you will test the **DragonSearch** function to see if it still works. Spoiler alert: it won't!

By now your sesssion may have expired, so lets grab a fresh one: (you should. be. in /lab6 path)

cd /home/ec2-user/environment/lab6

node ../lab5/solution/login.js test dave@dragoncardgame001.com apple

Then use that session instead of the <FMI> in the command below.

1. This command will test the Lambda function usinbg the CLI passing in a payload, similar to the way you have been testing things in the lambda console. REPLACE THE <FMI> with your new session\_id\_str above.
2. aws lambda invoke --function-name DragonSearch --payload '{"user\_name\_str": "davey65", "session\_id\_str": "<FMI>", "dragon\_name\_str": "Dexler"}' lambda-output.json && cat lambda-output.json | jq

You should see an output that looks like the following:

{

"ExecutedVersion": "$LATEST",

"StatusCode": 200

}

[

{

"family": {

"S": "green"

},

"damage": {

"N": "4"

},

"description": {

"S": "Dexler is a protector of the earth and forests. He is as green as the earth and burrows into the ground for protection and extra defense."

},

"protection": {

"N": "2"

},

"dragon\_name": {

"S": "Dexler"

}

}

]

1. Now that you know that your Lambda function can communicate with DynamoDB while not running from within the VPC, you need to lock this down. You will need to configure the VPC resources first. Execute the following command to create the resources:
2. Enable the script to be executed with this,

You should still be in the */lab6* path.

1. chmod +x resources/create\_vpc\_resources.sh
2. Now run
3. ./resources/create\_vpc\_resources.sh

The output should be similar to the following:

Creating VPC...

VPC ID 'vpc-0675ae344c5dd885e' CREATED.

Main Route Table ID is 'rtb-08761a962ab5495c0'.

Creating Private Subnet...

Subnet ID 'subnet-0048dd3333f83d25f' CREATED.

Creating Security Group for Lambda...

Lambda Security Group ID 'sg-047d1c2fee7131ffc' CREATED.

COMPLETED

⚠️ Take a note of this output as it will be used in the in the next tasks.

All of the resources are created and you are ready to modify the **DragonSearch** Lambda function so it uses the VPC instead.

1. Execute the following command to do that. You will need to replace the <FMI>s. Namely the Subnet ID from the previous command (eg. subnet-018e23...). You will also need to replace the <SECURITY GROUP ID> with the **Lambda Security Group ID** from the previous command (eg. sg-001453...).
2. aws lambda update-function-configuration --function-name DragonSearch --vpc-config SubnetIds=<FMI>,SecurityGroupIds=<FMI>

You should see a JSON description of your Lambda function similar to the following:

{

"FunctionName": "DragonSearch",

"LastModified": "2019-06-12T17:34:55.424+0000",

"RevisionId": "5d6a4484-bd03-4317-ba34-c61b0d46d82f",

"MemorySize": 128,

"Version": "$LATEST",

"Role": "arn:aws:iam::xxxxxxxxxxxxx:role/edx-ddb-dragonsearch-role",

"Timeout": 10,

"Runtime": "nodejs10.x",

"TracingConfig": {

"Mode": "Active"

},

"CodeSha256": "BzbaDDB0M+ednVCezWkb25xgShWUqY6DGJxuJSHGDSHD54=",

"Description": "",

"VpcConfig": {

"SubnetIds": [

"subnet-0048dd3333f83d25f"

],

"VpcId": "vpc-0675ae344c5dd885e",

"SecurityGroupIds": [

"sg-047d1c2fee7131ffc"

]

},

"CodeSize": 1440776,

"FunctionArn": "arn:aws:lambda:us-east-1:xxxxxxxxx:function:DragonSearch",

"Handler": "index.handler"

}

Test the Lambda function to see if you can communicate with DynamoDB now by running the same command as previously. Note that it could take up to 10 seconds for getting the output. That is normal as your Lambda function needs to timeout. However by now your sesssion may have expired, so lets grab a fresh one first.

node ../lab5/solution/login.js test dave@dragoncardgame001.com apple

Use that session\_id\_str here (repalce the <FMI>:

aws lambda invoke --function-name DragonSearch --payload '{"user\_name\_str": "davey65", "session\_id\_str": "<FMI>", "dragon\_name\_str": "Dexler"}' lambda-output.json && cat lambda-output.json | jq

You should see an output that looks like the following:

{

"FunctionError": "Unhandled",

"ExecutedVersion": "$LATEST",

"StatusCode": 200

}

{

"errorMessage": "2019-06-12T17:39:27.752Z 74d9c774-4380-4b54-b092-45e180075020 Task timed out after 10.01 seconds"

}

It sounds like your Lambda function doesn't work anymore, yet all you did was to modify it to run within the VPC. So this means it has to be an issue with the communication with DynamoDB. This is what we woudl expect as normlaly lambda woudl comunictae over the interenet to dyanmo db endpoiint. If it sindie a VPC it does not have by default access to reaxch the internet.

Looks like we are too protected now, we need to allow access to DynamoDB from calls made within the VPC (ie where our new lambda function is sitting).

Real world tip: To really lock it down, you would also prevent standard (none VPC) calls from interacting with DynamoDB. However for purposes of the lab we are not doing that right now.

## Step 4: Create the VPC Gateway Endpoint for DynamoDB

In this task, you will first create the **VPC Gateway Endpoint** for DynamoDB by passing the **VPC ID** and **Route Table** you created from the previous step.

You will then test the **DragonSearch** Lambda function to see if it can now communicate with DynamoDB from within the VPC. It should work with no code changes, it's all done via networking.

Finally, you will test everything in the website.

1. To create the VPC Gateway Endpoint, execute the following command. You will need to replace the <VPC ID> with the VPC ID from the previous task (eg. vpc-0831c0...). You will also need to replace the <ROUTE TABLE ID> with the Main Route Table ID from the previous task (eg. rtb-0fdc313...).
2. aws ec2 create-vpc-endpoint --service-name com.amazonaws.us-east-1.dynamodb --vpc-id <VPC ID> --route-table-ids <ROUTE TABLE ID>

The output of the command should be similar to the following:

{

"VpcEndpoint": {

"PolicyDocument": "{\"Version\":\"2008-10-17\",\"Statement\":[{\"Effect\":\"Allow\",\"Principal\":\"\*\",\"Action\":\"\*\",\"Resource\":\"\*\"}]}",

"VpcId": "vpc-0675ae344c5dd885e",

"NetworkInterfaceIds": [],

"SubnetIds": [],

"PrivateDnsEnabled": false,

"State": "available",

"ServiceName": "com.amazonaws.us-east-1.dynamodb",

"RouteTableIds": [

"rtb-08761a962ab5495c0"

],

"Groups": [],

"VpcEndpointId": "vpce-07d476773a2401268",

"VpcEndpointType": "Gateway",

"CreationTimestamp": "2019-06-12T17:48:50.000Z",

"DnsEntries": []

}

}

You can see that a Policy Document has been created automatically that allows everyone "Principal":"\*" inside the VPC to do any action "Action":"\*" against any DynamoDB resources "Resource":"\*". So if you wanted to only allow certain actions or resources through the VPC Gateway Endpoint, you could modify this Policy Document.

1. Now that the VPC Gateway Endpoint for DynamoDB is created, it's time to test the **DragonSearch** Lambda function the same way as you did in the previous task by running the following command.
2. Grab a session if you need one

node ../lab5/solution/login.js test dave@dragoncardgame001.com apple

Use that session\_id\_str here (repalce the <FMI>:

aws lambda invoke --function-name DragonSearch --payload '{"user\_name\_str": "davey65", "session\_id\_str": "<FMI>", "dragon\_name\_str": "Dexler"}' lambda-output.json && cat lambda-output.json | jq

You should see an output exactly the same as the last time this test worked. It should look like the following:

{

"ExecutedVersion": "$LATEST",

"StatusCode": 200

}

[

{

"family": {

"S": "green"

},

"damage": {

"N": "4"

},

"description": {

"S": "Dexler is a protector of the earth and forests. He is as green as the earth and burrows into the ground for protection and extra defense."

},

"protection": {

"N": "2"

},

"dragon\_name": {

"S": "Dexler"

}

}

]

Head over to your website (index3.html) Using your bucket name

http://<FMI>.s3-website-us-east-1.amazonaws.com/index3.html

Log in as dave

dave@dragoncardgame001.com

apple

You should see a list of Dragons as if nothing was done. Yet our lambda function is sat within our VPC to keep Steve happy .

## Step 5: Remove the Lambda function from the VPC -optional

1. If you ever want to remove your lambda function from the VPC you can run this.
2. aws lambda update-function-configuration --function-name DragonSearch --vpc-config SubnetIds=[],SecurityGroupIds=[]

**Congratulations!** You have completed exercise 6.

## Exercise 7

[version\_1.0.2]

# Exercise: Edit items on Amazon DynamoDB using the AWS Software Development Kit (AWS SDK)

In this exercise, you will learn how to develop with Amazon DynamoDB by using the AWS Software Development Kit (AWS SDK). Following the scenario provided, you will create administration features, using a DynamoDB table and using the AWS SDK. This exercise gives you hands-on experience with both Amazon DynamoDB and AWS Cloud9.

## Objectives

After completing this exercise, you will be able to use the AWS SDKs to do the following:

* Add **new attributes** to item a table, demonstrating the flexible schema
* **Edit** existing item attributes.
* Use **conditional** updates.
* Use **transactions**.

## Story continued

Mary is very happy with the security of the card data, and wants you to add a feature that only allows her to update card data.

So you decide that you will create entry in the user table for Mary, and using DynamoDB's flexible schema add an admin (Boolean) attribute setting it to true just for her.

You figure that when Mary logs in it would be cool if you could add an admin\_boo attribute to the sessions table, so the front end website can identify Mary as an admin and show the editing features just for her, which saves you creating a separate page in the website just for administration.

## Activity summary

* Create a new user (Mary) and flag her as a admin create\_mary\_admin.js.
* edit\_card.js allows admins to update items if they pass in some new attributes for a dragon.
* Add API GW path /edit to point to it.
* Edit the login function from lab5 to allow a special session for admins.
* Log into site as Mary to see edit options.
* Protect data and keep it in scope, using conditions and application code.
* Update the dragon power table at the same time (**transactions**) to keep our data in sync.

## Prepare the exercise

Before you can start this exercise, you need to import some files and install some modules in the AWS Cloud9 environment that has been prepared for you.

1. From the AWS Management Console, go to the **Services** menu and choose **Cloud9**.
2. Choose **Open IDE** to open the AWS Cloud9 environment.
3. To get the files that will be used for this exercise, go to the Cloud9 **bash terminal** (at the bottom of the page) and run the following wget command:

wget https://s3.amazonaws.com/awsu-hosting/edx\_dynamo/c9/dynamo-admin/lab7.zip -P /home/ec2-user/environment

You should also see that a root folder called **dynamolab** with a lab7.zip file has been downloaded and added to your AWS Cloud9 filesystem (on the top left).

1. To unzip the lab7.zip file, by running the following command:

unzip lab7.zip

This may take a few moments. In your Cloud9 filesystem.

1. To keep things clean, run the following commands to remove the zip file:

rm lab7.zip && cd lab7

1. Select the black arrow next to the lab7 folder (top left) to expand it. Notice inside this lab7 folder there is a solution folder. **Try not to peek at the solution unless you really get stuck. Always TRY to code first.**

## Step 1: Create an admin user (Mary)

In order to edit items the user must be an admin. Currently only Mary is an admin, however she is not in the user database, and furthermore we have no way of identifying Mary from other users. We could check specifically for her user\_name in the website JavaScript however it is better to flag her as an admin in case later on she decides to give admin rights to another user.

1. Open the SDK docs and find the method for creating new items. Find out the correct method names and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#putItem-property>. |

You should be in your Cloud 9 environment.

You know how to add items to a table as you have done this before, however this time you will need to create a sparse attribute. Meaning there will be an attribute of admin (a Boolean value) that only appears as an attribute on Mary's item that you are about to create. All other users have the absence of this attribute rather than setting it to false, and thus saving storage space and using extra write capacity units.

One challenge here is that you created a searchable index on the table that is being used during logins.

This index email\_index was created before you thought about the idea of an admin attribute.

You might have thought you could simply update the existing index to accommodate the new projection, however that is not currently possible. The only actions you can perform on an existing GSI are modifying the WCUs and RCSUs. So the solution to this problem is to create a new index email\_admin\_index and delete the old one email\_index and update our login code with the new index name.

Instead of using the SDK (as you already created an index before with the AWS-SDK), you are going to do this via the console. It is very easy.

1. Choose **Services** and search for **DynamoDB**.
2. Choose **Tables** and choose **users**.
3. Choose **Indexes**.
4. Choose **Create index**.
5. Make sure to type **admin** and choose **Add** and type **password** and choose **Add**.
6. Make sure you have email\_admin\_index (all underscores)
7. Choose **Create index**.

You should wait until the index is built before deleting the old one.

It can take up to **5 minutes**. (Press refresh in the console). Don not start the next step until this index shows ACTIVE.

**Now it's time to write some code that creates a new user called Mary.**

1. Open up the create\_mary\_admin.js file inside the lab7 folder by double clicking on it.
2. Have the SDK docs open (as above) to help you
   1. Replace the <FMI> sections of the code in that file, so that the code creates a new user in the user table mary, along with an admin Boolean attribute set to true.
   2. Her email is mary@dragoncardgame001.com and pear for password, and her username is mary001 with her name as lowercase mary.
3. Save the file.
4. Go to the terminal and run your file using the respective run command below

node create\_mary\_admin.js

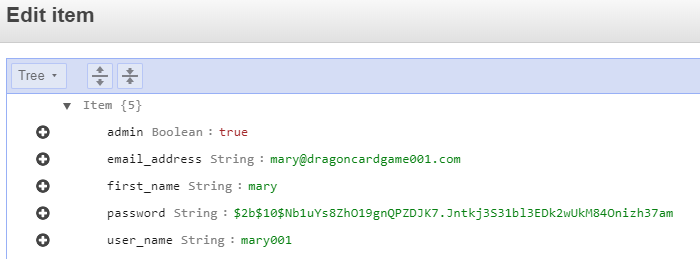
**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

## Confirm that your code worked.

You should see something like this in the console.

null { ConsumedCapacity: { TableName: 'users', CapacityUnits: 3 } }

1. If you go back to **DynamoDB** and choose the **users** table.
2. Choose **Items**.
3. You will see under **user\_name** that **mary001** was created.
4. If you choose **mary001** you will see the options we chose before.



###### Notes (real world gotcha):

Every time you run a script like this using putItem it will overwrite the old item completely. Meaning it will add the specified attributes and remove any attributes not specified. If later on, you decide to add more admin users remember to use updateItem (see later) not putItem, otherwise, allowing Dave to be an admin user would overwrite his password!

## Step 2: Create a specialized session for admins

When Mary logs in we should add an attribute to the sessions table differentiating her from other active sessions. This will make it easier for the front end and any backend edit functionality to know when to display and allow the editing of items.

**Time to write some code that modifies our login function LoginEdXDragonGame to add an admin flag to a session if they are actual administrators logging in (I.e. if it's Mary).**

1. Open up the updated\_login.js file inside the lab7 folder by double clicking on it.
2. Have the SDK docs open (as above) to help you
   1. Replace the <FMI> sections of the code in that file, so that the code creates an admin attribute on the session item created when an admin user logs in.
   2. Remember to update the code to point to the new index you just created email\_admin\_index, otherwise the admin flag will not be returned and picked up by the code in order to write the session as an admin session.
3. Save the file.
4. Go to the terminal and run your file using the respective run command below, i.e testing logging in with mary

node updated\_login.js test mary@dragoncardgame001.com pears

**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

## Confirm that your code worked.

You should see something like this in the console. (note theadmin\_boo key being returned)

Local test to log in a user with email of mary@dragoncardgame001.com

$2b$10$YZQBgIQzfeQeGsOymm1j5.RoKxUlKKgy7Z78.9Rt/M7BXg1tWn5GO pears

Password is correct

mary001 is an admin

{ ConsumedCapacity: { TableName: 'sessions', CapacityUnits: 1 } }

AWAITED b908d5e5-401a-4b33-8d47-ed6cfbc66c8a

null { user\_name\_str: 'mary001',

 session\_id\_str: 'b908d5e5-401a-4b33-8d47-ed6cfbc66c8a',

 admin\_boo: true }

This tells you Mary is now logged in and a session should have been created with an admin flag.

Run this command a few times, then check the sessions table. You will see Mary has multiple logins. which is ok, as she may be using multiple devices.

**NOTE** Real world tip: If you want to prevent an item from appearing more than once. i,e so that she could only have 1 session at a time this can be done with conditions.

###### From the AWS documentation:

To prevent a new item from replacing an existing item, use a conditional expression that contains the attribute\_not\_exists function with the name of the attribute being used as the partition key for the table. Since every record must contain that attribute, the attribute\_not\_exists function will only succeed if no matching item exists.

Anyway, next try logging in with Dave and compare the sessions in the DynamoDB console, you won't see an admin flag for Dave but he still has a valid login.

node updated\_login.js test dave@dragoncardgame001.com apple

You should see something like: (note **not** showing the admin\_boo key).

Local test to log in a user with email of dave@dragoncardgame001.com

$2b$10$TmOAsRPkSK/T2c2Vd9oKV.h5MAdflEu7alUG8sxaYe8SKiWBBW2n2 apple

Password is correct

{ ConsumedCapacity: { TableName: 'sessions', CapacityUnits: 1 } }

AWAITED f3a68c87-0ec0-41fe-8373-e8d8eca0e2b8

null { user\_name\_str: 'davey65',

session\_id\_str: 'f3a68c87-0ec0-41fe-8373-e8d8eca0e2b8' }

This is what will be used in the front end to display editing features, and be validated against at the back end when editing requests are made.

## Step 3: Move the test code to lambda to replace the site login

We need to update our Lambda function with this new code.

This step is simple, you just overwrite the lambda function LoginEdXDragonGame with the code you just created in updated\_login.js

1. Go back to the Lambda console through **Services** and search for **Lambda**.
2. Choose the **LoginEdXDragonGame** function.
3. Replace the contents of **login.js** with the code from **updated\_login.js**.
4. Choose **Save**.
5. Use the test case you created in a prior lab called fakePassword, which sends this as the payload.
6. {
7. "email\_address\_str": "dave@dragoncardgame001.com",
8. "attempted\_password\_str": "spaceship"
9. }

You should see something like this:

{

"errorMessage": "password does not match email"

}

Now create a new test case for Mary called maryLogin:

{

"email\_address\_str": "mary@dragoncardgame001.com",

"attempted\_password\_str": "pears"

}

You should see something like this:

{

"user\_name\_str": "mary001",

"session\_id\_str": "bdcd5a0d-8a1a-4f06-bce0-0411ac23b63f",

"admin\_boo": true

}

Now we know that our Lambda function works, we could go and test it at the API Gateway too, however it is probably easier to test the API from the website.

1. Go to index4.htmlAnd log in as Dave if you are not logged in already
2. dave@dragoncardgame001.com and apple

You will not see any edit functionality (as Dave is not an admin).

1. Logout (press logout davey65) and log back in as Mary.
2. mary@dragoncardgame001.com and pears

You will notice there is an edit icon on the top right of each card, showing that the website is recognizing Mary as having edit privileges.

If you click on this icon, the card should show you the back of the card where you can edit the title and text (by clicking on the text and typing over it) and adjust the damage and protection.

However when you click save icon, nothing happens. This is to be expected as we have not created the editing functionality at the back end yet.

## Step 4: Updating existing items

Up to this point, when you have wanted to change an item you would have simply overwritten it, replacing the old one using putItem. However you need to keep some attributes "as-is" and perhaps just change one of more of the item's attributes without affecting the other attributes on that item.

The website (index4.html) is pre-wired to send one or more updated attributes to your API (/edit) when you edit a card like you just tried to do.

You will create the API now.

Creating a new resource in your API Gateway endpoint is pretty easy as you have done something similar to this before for /login. You will need to follow similar steps again for creating /edit that will point to a function that we are going to call editCard.

1. Open the SDK docs and find the method for updating existing items. Find out the correct method names and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#updateItem-property> |

**Time to write some code that updates an existing item in the dragon\_stats table.**

1. Open up the edit\_1.js file inside the lab7 folder by double clicking on it.
2. Have the SDK docs open (as above) to help you
   1. Replace the <FMI> sections of the code in that file, so that the code updates an item in the dragon\_stats table.

We are not planning to update the location based keys (as they do not appear in the card website and will only be used during the game, which you thankfully are not creating).

We are also not going to edit the family key, we could, but it will mess up the card counts as we have a certain about of cards in a family per deck, and besides our images would no longer match up.

Our code needs to be flexible enough to accommodate one or more changes without overwriting existing attributes, and thus based upon what is passed in, we construct the update command in the code.

1. Once you have coded edit\_1.js, save the file.
2. Before you can run this line, you will need to create a session for Mary. You recently created a few sessions, however since you have been coding they may have expired due to the 20 minute Time To Live feature. Luckily creating a new session is easy, you can do this using this command just like before.
3. node updated\_login.js test mary@dragoncardgame001.com pears

Make a note of the session\_id\_str that was returned as you will need it in the next command.

Password is correct

mary001 is an admin

{ ConsumedCapacity: { TableName: 'sessions', CapacityUnits: 1 } }

AWAITED 636c5fdc-b168-42bf-9aa2-f80e5fd33b5a

null { user\_name\_str: 'mary001',

session\_id\_str: '36d03b18-545e-43fc-b0f6-6071f47ec8d4',

admin\_boo: true }

We wish to update the Dragon called Frost, so we are using a file we have put in your resources folder for lab 7, called update\_to\_frost.json

If you open update\_to\_frost you will see it simple contains the following:

{

"protection\_int": 6

}

Swap the <FMI> below with your session\_id\_str and run the update.

node edit\_1.js test mary001 <FMI> Frost

You should see something like this:

mary001 6172c938-184e-461f-9e0b-382137c334d4

match

SET protection = :protection,

{ Key: { dragon\_name: { S: 'Frost' } },

ExpressionAttributeValues: { ':protection': { N: '6' } },

UpdateExpression: 'SET protection = :protection',

ReturnValues: 'UPDATED\_NEW',

TableName: 'dragon\_stats' }

{ Attributes: { protection: { N: '6' } } }

null []

Time to check that the protection update to the dragon called Frost worked.

Select Frost in the drop down on the website and you will notice its (green circle) protection value of 3 changed to 6. He got new dragon armor from his friends in Lanzu ;). Woot!

Now try editing Atlasand renaming him to Atlantis.

First look at updates\_to\_atlas.json in your lab 7 resources folder, you will see that this time we are trying to make multiple changes to multiple attributes.

The file contains the following:

{

"damage\_int": 3,

"description\_str": "Loves drinking milk",

"dragon\_name\_str": "Atlantis",

"protection\_int": 4

}

The interesting thing here is that the dragon name is attempting to be altered, but it is a primary key!.

First try and run it, and see what happens. Get a new session (if you need to):

node updated\_login.js test mary@dragoncardgame001.com pears

node edit\_1.js test mary001 <FMI> Atlas

Note that this one fails all the attempted updates because we are using dragon\_name as a primary key (as per this message):

ValidationException: One or more parameter values were invalid: Cannot update attribute dragon\_name. This attribute is part of the key

Therefore we need to modify our code further to allow us to handle the special case of changing the dragon name.. We can't simply update the primary key in DynamoDB, so we need a different approach.

## Step 5: Editing the primary key

In this section you will write code that will check to see if they are trying to change a dragon name.

If that is the case, we will create a new item (dragon) using all the existing information merged with the new changes (if any) from the other attributes also potentially being altered.

Then finally delete the old record.

So it seems a bit long winded when all you want to do is change the dragon name, however we used that as our primary key so our only option is to create a new dragon with all the attributes its needs, then kill (slay) the old one.

The challenge here is that you could easily end up out of sync. Imagine if you added a new dragon and you were about to complete the second part and delete the old one, then something went wrong!

Meaning that last part failed for some reason (of which there could be many).

You would have 2 essentially identical dragons, just with different names.

So to fix this problem and to keep our tables and items all in sync, we will do all of this inside what we call a transaction. Think "banking transactions", where money comes out of one account and then is deposited into another, BOTH must work inside the banking transaction or it should FAIL.

So how do we do a transaction in this situation?

We could do this in code, and handle rollbacks but that's error prone, and kind of messy.

#### Luckily we can use a feature of DynamoDB called suprise-suprise - "**Transactions**".

To keep our tables and items all in sync, we will do all of this in a transaction. So it either all works or all fails, we don't want some entries calling the dragon Atlas and some others Atlantis. That would be messy.

1. Open the SDK docs and find the method for transactions. Find out the correct method names and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#transactWriteItems-property> |

**Time to write some code that allows you to change a dragon's name.**

1. Open up the edit\_2.js file inside the lab7 folder by double clicking on it.
2. Edit the file and replace the <FMI>s like normal.
3. Save the file.

Ensure you have a valid session first.

node updated\_login.js test mary@dragoncardgame001.com pears

Then use the session id to make your calls

Local test to log in a user with email of mary@dragoncardgame001.com

$2b$10$xAv4kjMWO9djWMU5B.7TEe0awWVZ3xKSxnL3u2QRCSR391fG6v.92 pears

Password is correct

mary001 is an admin

{ ConsumedCapacity: { TableName: 'sessions', CapacityUnits: 1 } }

AWAITED acce49d5-a16c-40fc-82f4-f752a11ff43f

null { user\_name\_str: 'mary001',

session\_id\_str: '6172c938-184e-461f-9e0b-382137c334d4',

admin\_boo: true }

Now try changing Atlas with your new enhanced code:

node edit\_2.js test mary001 <FMI> Atlas

This should work and show you:

mary001 6172c938-184e-461f-9e0b-382137c334d4

match

special case we need a transaction here

null 'wow that transaction worked'

Head to the DynamoDB console and under dragon\_stats do a scan. You will no longer see an entry for Atlas. However you will see an entry for Atlantis.

You could even head over to the website to see that change.

Head to the website and select Atlas.

You will see:

No dragon called Atlas found

Click ALL and scroll down to see Atlantis. (without a picture as unlike Sprinkles our celebrity dragon, he is camera shy).

Atlas was deleted and Atlantis created all in 1 transaction.

Congrats.

Job done right?

Nope there is another problem!

**If you try and update (and don't do this yet btw) an existing dragons name to a new dragons name that already exists it will overwrite the other dragon!**

This is not good. So we need to enforce that the new "proposed" dragon name cannot be one that is use already.

## Step 6: Conditions

We need to find a way to ensure that updates that attempt to create a negative protection value that are not allowed.

We could write code that searches for any dragons called the proposed name, and then say "nope", however there is better way.

We can enforce a "condition" on a DynamoDB table.

We can say in pseudocode:

CREATE dragon WHERE dragon\_name NOT EXIST

* Open the SDK docs and find the method for conditions. Find out the correct method names and establish what parameters you need to pass in.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#putItem-property> (study the conditions section) |

**Time to write some code that prevents overwriting of existing dragons.**

1. Open up the edit\_3.js file inside the lab7 folder by double clicking on it.
2. Edit the file and replace the <FMI>s like normal.
3. Save the file
4. Ensure you have a valid session first.
5. node updated\_login.js test mary@dragoncardgame001.com pears

Then use the session id to make your calls

In update\_to\_fireball.jsonyou will see that it try's to change Fireball's name to Blackhole, however we have a dragon already called Blackhole, so this is a good test case to check our code with.

We want this update to fail

{

"description\_str": "Always hiding in shadows",

"dragon\_name\_str": "Blackhole"

}

Once you have written your code (filled in the FMIs) save your file

1. Try this and hope it **fails** ;)
2. node edit\_3.js test mary001 <FMI> Fireball

If you see this **error**, you have done it correctly

$ node edit\_3.js test mary001 e6e74f94-6ff8-4b73-9f8e-7dcbfd6d05c5 Fireball

mary001 e6e74f94-6ff8-4b73-9f8e-7dcbfd6d05c5

match

special case we need a transaction here

TransactionCanceledException: Transaction cancelled, please refer cancellation reasons for specific reasons [None, ConditionalCheckFailed]

at Request.extractError (/home/ec2-user/node\_modules/aws-sdk/lib/protocol/json.js:51:27)

at Request.callListeners (/home/ec2-user/node\_modules/aws-sdk/lib/sequential\_executor.js:106:20)

at Request.emit (/home/ec2-user/node\_modules/aws-sdk/lib/sequential\_executor.js:78:10)

at Request.emit (/home/ec2-user/node\_modules/aws-sdk/lib/request.js:683:14)

at Request.transition (/home/ec2-user/node\_modules/aws-sdk/lib/request.js:22:10)

at AcceptorStateMachine.runTo (/home/ec2-user/node\_modules/aws-sdk/lib/state\_machine.js:14:12)

at /home/ec2-user/node\_modules/aws-sdk/lib/state\_machine.js:26:10

at Request.<anonymous> (/home/ec2-user/node\_modules/aws-sdk/lib/request.js:38:9)

at Request.<anonymous> (/home/ec2-user/node\_modules/aws-sdk/lib/request.js:685:12)

at Request.callListeners (/home/ec2-user/node\_modules/aws-sdk/lib/sequential\_executor.js:116:18)

message: 'Transaction cancelled, please refer cancellation reasons for specific reasons [None, ConditionalCheckFailed]',

code: 'TransactionCanceledException',

time: 2019-06-04T18:55:21.035Z,

requestId: 'F3S3TQ0DDC79CPHNH4532SQSBFVV4KQNSO5AEMVJF66Q9ASUAAJG',

statusCode: 400,

retryable: false,

retryDelay: 39.73114870214614 } null

**Congrats**

No more "Double Dragon".

However there is one other thing that you need to prevent. Damage and Protection values cannot be negative or over 10.

## Step 7: Code Validation

You might think that adding this bit of validation is easy, right?

You want to say only update this dragon's protection value if that value falls between 0 and 10.

No can do! Let me explain why:

You might think you could do something like this: (Again in pseudocode)

--update-expression "SET protection = :protection"

--condition-expression "protection < 10"

However that will not work on our situation.

To explain why this won't work let's look at shopping cart example, coming away from dragons just for a second.

We could say to the Database; if the price of an item is already at the lowest possible discounted price, you are not to discount it further.

E.g

The following example performs an UpdateItem operation. It attempts to reduce the Price of a product by 75—but the condition expression prevents the update if the current Price is below 500:

--update-expression "SET Price = Price - :discount"

--condition-expression "Price > :limit"

The key takeaway here is that it is the **existing** Price.

So why do we mention this and what has this got to do with dragons?

Well, if we attempted to do a **condition** on the protection value, and told our table to reject any value less than 0, it wouldn't work.

This is because at the time you are updating the item to the invalid value of say -8the protection value is VALID.

So what happens is, it will update it to -8 no problem!

See the issue? You now have an invalid value in your table. It didn't prevent it from being written.

To make things worse, further updates to correct it, will always FAIL, because the **current** protected value is now -8 and out if the range of the constraints, preventing any further updates!

So unlike the shopping cart example above which is kind of useful, we can't enforce the dragon protection or damage values to be constrained to the "proposed value".

We can't say "update this protection value if your proposed value is > 0 and < 10."

I mean we can, but it has to be in the application code, and not at the database constraint layer.

So let's add that in code anyway, as we don't want negative damage values for dragons, as they would go around healing everyone they breathed on

Interestingly, you actually have the same issue with S3 updating as you do with DynamoDB. If you want to change a dragon image like Castral.png to Funky.png, you have to create a copy of Castral.png first and call it Funky.png then delete the old one.

So we will add code for that too while we are in there.

Lab note: You won't have to write the S3 code, just add the bucket name where you see the FMI.

That way you have no more camera shy dragons, when you change their names on the website ;)

Remember when we added S3 permissions earlier in the course to the role we use for our lambda functions, well this was for this. #you're\_welcome

**Time to write some code that protects values outside of 0 to 10, and updates S3 if you change the dragon name.**

1. Open up the edit\_4.js file inside the lab7 folder by double clicking on it.
2. Edit the file and replace the <FMI>s like normal.
3. Save the file
4. Ensure you have a valid session first.
5. node updated\_login.js test mary@dragoncardgame001.com pears

Then use the session id to make your calls

Local test to log in a user with email of mary@dragoncardgame001.com

$2b$10$xAv4kjMWO9djWMU5B.7TEe0awWVZ3xKSxnL3u2QRCSR391fG6v.92 pears

Password is correct

mary001 is an admin

{ ConsumedCapacity: { TableName: 'sessions', CapacityUnits: 1 } }

AWAITED acce49d5-a16c-40fc-82f4-f752a11ff43f

null { user\_name\_str: 'mary001',

session\_id\_str: '88d12dda-5152-419f-8a2c-03ec01f5bd52',

admin\_boo: true }

Have a look at update\_to\_castral.json it contains the following:

{

"damage\_int": 3,

"protection\_int": -4

}

Again we hope this **fails** as -4 is not a valid value for protection..

node edit\_4.js test mary001 <FMI> Castral

You should see the following, indicating only the damage value was updated and the protection value failed and did not update due to our code validation.

mary001 01172a52-a338-44e1-94aa-9affbf15254d

match

SET damage = :damage,

{ Attributes: { damage: { N: '3' } } }

null { damage: { N: '3' } }

Notice that the protection was not updated. **Woot!**

Now try it with a valid value and changing the **name**, just to make sure it all works and also updates the **image** on s3.

Use update\_to\_dexler.json

{

"damage\_int": 8,

"description\_str": "Loves drinking black tea",

"dragon\_name\_str": "Firestorm",

"protection\_int": 4

}

node edit\_4.js test mary001 <FMI> Dexler

This should FAIL, because we have a dragin called "Firestorm".

mary001 93d0af7e-6d6f-4b91-9b4e-e50da5871946

match

special case we need a transaction here

{ TransactionCanceledException: Transaction cancelled, please refer cancellation reasons for specific reasons [None, ConditionalCheckFailed]

at Request.extractError (/home/ec2-user/node\_modules/aws-sdk/lib/protocol/json.js:51:27)

at Request.callListeners (/home/ec2-user/node\_modules/aws-sdk/lib/sequential\_executor.js:106:20)

at Request.emit (/home/ec2-user/node\_modules/aws-sdk/lib/sequential\_executor.js:78:10)

at Request.emit (/home/ec2-user/node\_modules/aws-sdk/lib/request.js:683:14)

at Request.transition (/home/ec2-user/node\_modules/aws-sdk/lib/request.js:22:10)

at AcceptorStateMachine.runTo (/home/ec2-user/node\_modules/aws-sdk/lib/state\_machine.js:14:12)

at /home/ec2-user/node\_modules/aws-sdk/lib/state\_machine.js:26:10

at Request.<anonymous> (/home/ec2-user/node\_modules/aws-sdk/lib/request.js:38:9)

at Request.<anonymous> (/home/ec2-user/node\_modules/aws-sdk/lib/request.js:685:12)

at Request.callListeners (/home/ec2-user/node\_modules/aws-sdk/lib/sequential\_executor.js:116:18)

message: 'Transaction cancelled, please refer cancellation reasons for specific reasons [None, ConditionalCheckFailed]',

code: 'TransactionCanceledException',

time: 2019-06-13T19:15:58.011Z,

requestId: '34CV9ES2C13UDLCMJ3MTETBM1FVV4KQNSO5AEMVJF66Q9ASUAAJG',

statusCode: 400,

retryable: false,

retryDelay: 37.351601045896324 } null

Finally lets try this one:

Use update\_to\_fireball.json

{

"description\_str": "Always hiding in shadows",

"dragon\_name\_str": "Sanguia"

}

mary001 93d0af7e-6d6f-4b91-9b4e-e50da5871946

match

special case we need a transaction here

swapping image

Sanguia Fireball

image changed

null 'ok'

null { location\_neighborhood: { S: 'morgan rd' },

damage: { N: '8' },

location\_city: { S: 'page' },

family: { S: 'blue' },

description: { S: 'Always hiding in shadows' },

protection: { N: '6' },

location\_country: { S: 'usa' },

location\_state: { S: 'arizona' },

dragon\_name: { S: 'Sanguia' } }

**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

**Congrats** Now we just need to update the API with you new editing code.

## Step 8: Final

All that is left to do is to create a new lambda function called editDragons so you can edit cards directly on the website.

1. Choose **Services** and search for **lambda**.
2. Choose **Create function**.
3. Under **Function name** type in editDragons.
4. Leave the **Runtime** as **Node.js 10.x**.
5. Under **Permissions** select **Choose or create an execution role**.
6. Choose the drop-down and select **Use an existing role**.
7. Choose the drop-down under **Existing role** and select call-dynamodb-role.
8. Choose **Create function**.
9. Replace the contents of **index.js** with the code from **edit\_4.js**.
10. Under **Basic settings** change the **Timeout** to 10 sec.
11. Enable **Xray**
12. Select NO VPC (for puposes of this lab)
13. Choose **Save**.

* Now head over to API Gateway, as you have already tested this code.

1. Choose **Services** and search for **api**.
2. Choose the **DragonSearchAPI**.
3. Select the / resource and choose **Actions** and **Create Resource**.
4. Under **Resource Name** type in edit and choose **Create Resource**.
5. Choose **Actions** and **Create Method**. Select **POST** and click the checkmark.
6. Leave **Integration type** as Lambda Function and in the **Lambda Function** box type in editDragons.
7. Choose **Save** and then **OK** at the add permission screen.
8. Select the new /edit resource and choose **Actions** and **Enable CORS**.
9. Select **DEFAULT 4XX** and **DEFAULT 5XX** and choose **Enable CORS and replace existing CORS headers**.
10. Choose **Yes, replace existing values**.
11. Choose **Actions** and then **Deploy API**
12. Select **prod** and choose **Deploy**

Refresh your s3 website (index4.html) You may need to log out and log back in again as Mary)

mary@dragoncardgame001.com

pears

Try changing a dragon name to an existing dragon name (it will fail and you will see no changes).

Try changing to a dragon (with or without changing other attributes) that does exist and you should see it change and not be camera shy (i.e it will have an image), and the old dragon will be gone.

Real world tip: In production you would have the website call up a list of dragons dynamically , instead of hard coded, and woudl update and name changes.

**Congrats you are all done.**

Mary plays with your new site and loves all the functionality you provided, and rewards you with some chocolate cake

...then why is there a lab 8 if you are all done

## Exercise 8

[version\_1.0.2]

# Exercise: Single table with Amazon DynamoDB using the AWS Software Development Kit (AWS SDK)

## Overview

In this exercise, you will learn how to develop with Amazon DynamoDB by using the AWS Software Development Kit (AWS SDK). Following the scenario provided, you will work with the single table concept of DynamoDB and use the AWS SDK. This lab gives you hands-on experience with both Amazon DynamoDB and AWS Cloud9.

## Objectives

After completing this lab, you will be able to use the AWS SDKs to do the following:

* Create a table for the dragon data using single table design.
* Uploading multiple items to a single table DynamoDB table.
* Do advanced querying using composite sort key

## Story continued

So your work with Mary is done. She is delighted with the work you have done for her, and her developers can now work towards creating the full game engine.

However you want to push your DynamoDB skills further by implementing a more advanced schema (single table). You wanted to see how this would affect the existing queries and if you could add more elaborate search functionality to the site.

As this is just going to be an experiment to show Mary the single table approach, you are not going to reimplement all the editing and password protected areas for this. You are simply going to stand on the shoulders of the older index2.html by creating a new experimental version (index5.html). That way you can prove to Mary that not only is the same functionality possible using a single table, but also that more advanced queries can be carried out.

To save time, you figure you will be able to leverage much of the boilerplate code you have used thus far and tweak it accordingly for your new API.

Your ideas are as follows:

* Create a single table version of dragon data .
* Ensure that the basic version of the site can maintain the same functionality using this single table pattern
* Prove to yourself, and Mary that there are real advantages in using the single table concept.

## Your thought process

##### You first reflect on the dragon queries that you have on the current (old index2.html) site, and think about how you can achieve the same thing with a single table schema.

What you had in index2.html:

* Ability to return all dragons filtering out all but a few attributes, using a scan and filter.
* Ability to return information on a specific dragon, again returning only a few attributes, using a query.

Since you built index2.html, you have learned (in lab 4) that a scan is **usually** less efficient that a query, so we will keep that in mind when we look at the single table version.

It is important that you can still do these types of query, so your first draft of the single table is as follows.

| **PK (PK)** | **Location** |  |
| --- | --- | --- |
|  | <Location\_country\_value>:<Location\_state\_value>:<Location\_city\_value>:<Location\_neighborhood\_value> | dragon\_name:, protection:, damage:, description:, family: |

So the biggest change here is that we have a new Primary Key called PK. It is going to be a random key of type String, with no sort key. Similar to how you did the session table earlier in lab 5.

Also note we are creating a location attribute as a composite key value of all the location attributes. This will reduce table size a little bit, and allow us to create GSIs (later on) that will enable advanced searching. (more on that stuff later).

So in pseudocode (not real code), you would probably do something conceptually a bit like this:

SCAN single\_dragon\_table --FILTER where dragon\_name EXISTS

This is not super efficient, but not really any worse that what you had before with the dragon\_stats scan.

However things start to fall down when you come to search for a specific dragon. It becomes pretty inefficient to use the "scan and filter" approach as you saw in lab 4.

You really want to use a query, but you can't...Well, not on this single\_dragon\_table at least.

So you think, "an Index to the rescue!"

Great idea!

You could create a GSI using dragon\_name as the primary key, with no sort key. You could call it dragon\_stats\_index, projecting only the things you want returning, such as description, protection and damage.

Note the primary key (PK) of this index (dragon\_name) will always be returned even when not specified in the projection

The dragon\_stats\_index would look like this:

| **PK** | **<attributes>** |
| --- | --- |
| <dragon\_name\_value> | protection:<value>, damage:<value>, description:<value>, family:<value> [uuid-PK added by default] |

Notice no location info bloating the index. #win

So in pseudocode, getting just one dragon would look a bit like this

QUERY dragon\_stats\_index WHERE dragon\_name (PK) = dragon\_requested

So you think using these 2 types of queries, you could achieve the same site functionality.

The advantage here is that the GSI is a sparse index, only keeping items that have a dragon\_name, so later on when you add more items to the single table that are not items with a dragon name attribute, it can efficiently pull only what it needs.

Let's think a little more about the concept of a scan (again in pseudocode)

SCAN single\_dragon\_table --FILTER where dragon\_name EXISTS

This actually would be slightly worse than the non single table version scan because when you add many items later it will scan all of the items, which is far from ideal.

If we move to the single table concept it needs to be better not worse! Otherwise what's the point.

The query we can do on the single table's index is really good, however the scan is not so good. But guess what? You already have the perfect scan opportunity, you simply scan the index!

So in pseudocode this is definitely the way to go.

SCAN dragon\_stats\_index

This is very efficient it only returns what you need, with no filtering required, other than having to ignore the PK which gets returned whether we like it or not. This scan doesn't touch any items that don't need returning. This is the best type of scan.

Ok, so now you know the plan of action to recreate the base site functionality, let's create our first attempt of a single table and build an API Gateway and the Lambda function behind it.

### Goals of this lab:

* Exercise 1) Create the single table (single\_dragon\_table) along with the GSI (dragon\_name\_index)
* Exercise 2) Seed the single table by uploading items via JSON (similar to what you did before)

NOTE, as you go through later parts of tis lab, you will add other items from the other tables into this single table. For now (Part 2), we are just concentrating on getting the dragon stats in there, to ensure we can replicate the base site functionality first .

As mentioned, we are also not going to integrate all the editing features we did in lab 7 or the membership stuff from lab 5, as at this stage we are simply proving that the same base functionality is possible (and slightly better) using the single table concept.

* Exercise 3) Create and test a new API and Lambda backend, that will talk to the single table's index.
* Exercise 4) Test the new website works in the same way (but more efficiently) that the old website.

## Prepare the lab

Before you can start this exercise, you need to import some files and install some modules in the AWS Cloud9 environment that has been prepared for you.

1. From your Cloud 9 Environment collapse any folders and close any tabs you are not longer using.
2. Ensure you are in the right path in your Cloud9 terminal using:

cd /home/ec2-user/environment

To get the files that will be used for this exercise, go to the Cloud 9 **bash terminal** (at the bottom of the page) and run the following wget command:

wget https://s3.amazonaws.com/awsu-hosting/edx\_dynamo/c9/dynamo-single/lab8.zip -P /home/ec2-user/environment

You should also see that a root folder called **dynamolab** with a lab8.zip file has been downloaded and added to your AWS Cloud9 filesystem (on the top left).

1. To unzip the lab8.zip file, by running the following command:

unzip lab8.zip

This may take a few moments. In your Cloud9 filesystem.

1. To keep things clean, run the following commands to remove the zip file:

rm lab8.zip && cd lab8

1. Select the black arrow next to the lab8 folder (top left) to expand it. Notice inside this lab7 folder there is a solution folder. **Try not to peek at the solution unless you really get stuck. Always TRY to code first.**

## Step 1: Create the single table (single\_dragon\_table)

Creating a table and creating an index shouldn't take you long because you have done this sort of thing before,

1. Open the SDK docs and find the method for creating a table and index.

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#createTable-property>. |

You should be in your Cloud 9 environment.

**It's time to write some code that creates a new table and index.**

1. Open up the create\_single\_table\_and\_index.js file inside the lab8 folder by double clicking on it.
2. Have the SDK docs open (as above) to help you
   1. Replace the sections of the code in that file, so that the code creates a new table called single\_dragon\_table along with an index called dragon\_stats\_index in us-east-1.

This is the structure we are going for:

| **PK** |  |
| --- | --- |
| <dragon\_name\_value> | protection:, damage:, description:, family: [uuid-PK added by default] |

1. Save the file.
2. Go to the terminal and run your file using the respective run command below
3. node create\_single\_table\_and\_index.js

**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

## Confirm that your code worked.

You should see something like this in the console.

null { TableDescription:

{ AttributeDefinitions: [ [Object], [Object] ],

TableName: 'single\_dragon\_table',

KeySchema: [ [Object] ],

TableStatus: 'CREATING',

CreationDateTime: 2019-05-30T15:13:06.336Z,

ProvisionedThroughput:

{ NumberOfDecreasesToday: 0,

ReadCapacityUnits: 0,

WriteCapacityUnits: 0 },

TableSizeBytes: 0,

ItemCount: 0,

TableArn: 'arn:aws:dynamodb:us-east-1:000000000000:table/single\_dragon\_table',

TableId: 'db146e33-d82f-4143-8103-fe45db8e0cca',

BillingModeSummary: { BillingMode: 'PAY\_PER\_REQUEST' },

GlobalSecondaryIndexes: [ [Object] ] } }

Head over to your DynamoDB console, and you will see that you have a new table called single\_table being created.

⚠️ Wait until its says ACTIVE before moving on, because you can't update a table when it is not ACTIVE, and we need to do that next. (This can take upto 5 minutes)

## Step 2: Seed the single table with dragon stats

Once your table is ACTIVE we can seed it like we did before.

The only different part here is that you will need to create a composite location attribute.

You are going to use this location attributes (later on) as part of a GSI where it will play the role of Sort Key, to enable some cool search functionality.

location\_country\_value#location\_state\_value#location\_city\_value#location\_neighborhood\_value

Written like so: (spaces are ok), all lowercase.

\*FYI: Location is where these dragons were last spotted roaming in our Universe by our **dragon drones**.

usa#nevada#las vegas#spring valley

1. Open the SDK docs and find the method for **batchWriteItem**

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#batchWriteItem-property>. |

You should be in your Cloud 9 environment.

**It's time to write some code that creates a new table and index.**

1. Open up the seed\_single\_table.js file inside the lab8 folder by double clicking on it.
2. Have the SDK docs open (as above) to help you
   1. Replace the sections of the code in that file, so that the code reads the items from the old `dragon\_stats JSON files you had in lab 3 (copied over to lab 8/resources, and use this to fill in the new single\_dragon\_table. You'll probably notice the code is very similar to what you had in lab 3, just with a different schema
3. Once done, save the file.
4. Go to the Cloud 9 terminal and run your file using the command below

⚠️ Make sure you don't run this following command twice, as the PK is a unique identifier (UUID) with negligible chance of collisions, so each time you run this command it will create new items over and over. You could add an *exists* condition (which we will talk about later) to prevent this, but it is ***just easier*** not to seed it twice ;)

node seed\_single\_table.js # just the once please ;)

**IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER**.

## Confirm that your code worked.

You should see something like this in the console.

[ { UnprocessedItems: {} }, { UnprocessedItems: {} } ]

HowFastWasThat: 89.387ms

* Check your DynamoDB console and press the **single\_dragon\_table**.
* Now press items. You should see items in there with UUID as the Primary Key.
* Where you see the word scan in the drop down next to it you can choose the Index dragon\_stats\_index. Then press **scan** and you will see the same items, except with only the projected attributes that you set for that index.

## Step 3: Create and test a new API resource along with a Lambda backend, that will talk to the single table's index

You will need to create a new API gateway resource here, as we don't want to interfere with the working website.

We will call this resource single

We have provided you with a clone of index2.html called index5.html with a few tweeks in the JavaScript, that you will use (later) to check your code worked.

I.e its basically the old base site without all the membership and editing features,

The first step is to create a new Lambda function that will connect to the dragon\_stats\_index instead of the old table/index. Therefore the code will look very similar as before.

1. Open the SDK docs and find the method for querying DynamoDB

| **Language** | **AWS documentation deep link** |
| --- | --- |
| NODE.JS (8.16.0) | <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#query-property> |

**It's time to write some code that queries your single tables index**.

1. Close down any Cloud 9 tabs you are no longer using.
2. Open up the index\_query.js file inside the lab8 folder by double clicking on it.
3. Have the SDK docs open (as above) to help you
   1. Replace the sections of the code in that file, so that the code queries from the new dragon\_stats\_index.
   2. For lab brevity you don't need to integrate Xray right now.
4. Save the file.
5. Go to the terminal and run your file using the respective run command below

node index\_query.js test "Fireball"

This shoudl trigger the justOneDragon method and you should see something like this:

Local test for a dragon called Fireball

null [ { family: { S: 'red' },

damage: { N: '2' },

description:

{ S: 'Fireball is a young dragon in training. He is learning how to control his fire, but is still lethal.' },

pk: { S: 'd5ec708a-65d0-423c-a7c6-8523bf5a0572' },

protection: { N: '6' },

dragon\_name: { S: 'Fireball' } } ]

Now try a scan (i.e find all dragons) - uses the *scanIndex* method

node index\_query.js test "All"

.......

....... (many records ^^^

{ family: { S: 'green' },

damage: { N: '3' },

description:

{ S: 'Jerichombur is a dragon of mischief. His earth crushing roar can be heard for miles.' },

pk: { S: '1e71d947-7d89-4a59-8272-0cc4fb39bfc9' },

protection: { N: '5' },

dragon\_name: { S: 'Jerichombur' } } ]

Total items scanned: 48

Did you notice how the code is much simpler than what you had before? No ExpressionAttributeNames required, no ProjectionExpression required. So this is another benefit of the single table concept and being smart with indexes. Less code… #winning

Real world tip: The only thing is, PK comes back too, but that is easy enough to ignore and is only a few bytes anyway. However you might be glad you got this returned anyway, as is allows you to potentially to do very fast item lookups using the table directly if the situation needs it.

We just need to create a lambda function out of this code that we just tested. So we will use the same code and steps (almost) as you did last time to publish the lambda function.

#### Step 3A): Create the new Lambda function with your new index\_query code

You already have your new and improved index\_query script ready and tested, so all you need to do now is create a Lambda function out of that code.

You are not creating this inside the VPC like in lab 6, as this is just an experimental proof of concept.

Once Mary and her team are all sold on the idea of the single table, they can always move it into the VPC and add XRAY (like in lab 4 and 6), along with all the other features such as membership and editing.

Your code in the index\_query script was set up to work both in a Cloud9 testing environment and also within a Lambda environment. You do not need to alter the code.

You will create a Lambda function, passing in that role you just created back in lab 2 called call-dynamodb-role, then paste in your code "as is" from your new index\_query script.

TIP: You would normally use the principle of least privilege and remove things like access to S3 and so forth, but this is just an experiment for you and Mary, so its just faster and easier for you to just use the role you already created earlier.

###### Follow these steps to publish your code as a lambda function:

1. Choose **services** and search for **lambda**.
2. Choose **lambda** from the drop-down list.
3. Choose **Create function**.
4. Type in ImprovedDragonSearch for the **Function name**.
5. Use **Node.js 10x** for the **Runtime**.
6. Under **Permissions** choose **Choose or create an execution role**.
7. Under **Execution role** choose **Use an existing role**.
8. In the **Existing role** drop-down choose the role we created above call-dynamodb-role.
9. Finally choose **Create function**. Ignore any warnings.
10. Copy and paste the code from your index\_query.js file into the code editor replacing the contents of index.js.
11. Under **Basic settings**. Change the timeout to 10 sec.
12. Choose **Save**.
13. Configure a test event called Fireball in the usual way

{

"dragon\_name\_str": "Fireball"

}

Under Details you shoud see:

[

{

"family": {

"S": "red"

},

"damage": {

"N": "2"

},

"description": {

"S": "Fireball is a young dragon in training. He is learning how to control his fire, but is still lethal."

},

"pk": {

"S": "d5ec708a-65d0-423c-a7c6-8523bf5a0572"

},

"protection": {

"N": "6"

},

"dragon\_name": {

"S": "Fireball"

}

}

]

1. Now try creating a new test event called All

{

"dragon\_name\_str": "All"

}

View the details section, and you should now see **all** the dragon info.

Remember we are not validating like before, where we protected the cards from valid sessions. This is just a proof of concept to show Mary the single table idea.

Next we need to create the **API Gateway** resource and point your new Lambda function.

#### Step 3B): Create a new resource in the DragonSearchAPI

1. Go to **API gateway** and select your **DragonSearchAPI**.
2. Select Resources
3. Select the / so it is highlighted.
4. Choose **Actions** and **Create Resource**.
5. Leave **CORS** and **configure as proxy** UNCHECKED.
6. Call the resource single (lowercase).
7. Choose **Create Resource**.
8. Select /single and choose **Actions** and **Create Method**.
9. Choose **POST** and click the checkmark.
10. Under Lambda Function type in imporvedDraagonSearch.
11. Set the **timeout** to 10000 seconds.
12. Choose **Save**.
13. Click **OK** to bypass the permissions pop-up.
14. Choose **TEST** and leave the **Request Body** blank.

You should see all dragons data returned (without location info).

Test it one more time with this in the **Request Body**:

{

"dragon\_name\_str": "Fireball"

}

You should see

[

{

"family": {

"S": "red"

},

"damage": {

"N": "2"

},

"description": {

"S": "Fireball is a young dragon in training. He is learning how to control his fire, but is still lethal."

},

"pk": {

"S": "d5ec708a-65d0-423c-a7c6-8523bf5a0572"

},

"protection": {

"N": "6"

},

"dragon\_name": {

"S": "Fireball"

}

}

]

1. Now select /single so it is highlighted and choose **Actions** and **Enable CORS**.
2. Select the **DEFAULT 4XX** and **DEFAULT 5XX** checkboxes and press **Enable CORS and replace existing CORS headers**.
3. Choose **Yes, replace existing values** (if that message appears).
4. Choose **Actions** and choose **Deploy API**.
5. Choose prod and choose **Deploy**.
6. Visit your S3 site and append index5.html to the end of the link.

Is the site is working as expected (without the login and editing features)?

If so, **congrats** you are now running the all the dragon queries using the improved single table version.

# PART 2 - The real single table concept

Ok, so this is great. You are able to show Mary a rough proof of concept without affecting what she already has with the old site.

Everything works the same (slightly more efficient this time though), but now its time to really experiment with the single table concept, as really all you have right now is a reconstructed version of the old dragon\_stats table. That was already single table really right ;)

To make this a **true** single table, we need to consolidate all the other dragon data tables into it, so we have 1 big table instead of many little ones. Which I am sure would be an easy sell when talking to Mary and her developers.

You also want to be sure that this concept will also allow them to query the dragon data in interesting ways, as that is what is really going to convince them to migrate to a single table.

#### So let's think about the schema again.

Here is what we have currently.

| **PK (PK)** | **Location** |  |
| --- | --- | --- |
|  | <Location\_country\_value>:<Location\_city\_value>:<Location\_neighborhood\_value> | dragon\_name:, protection:, damage:, description:, family: |

On the live site we have a total of 4 tables with dragon data. However one of the tables is the odd one out. It is the one called current\_power\_table and it's going to be an in-game session table.

It looks a bit like this:

| **game\_id (PK)** | **dragon\_name** | **current\_will\_not\_fight\_credits** | **current\_endurance (dynamic)** |
| --- | --- | --- | --- |
| 56syjdh8756 | Cassidiuma | 2 | 8 |

This table is different that the others that contain actual dragon data. This is a table of in-game progress and is designed to keep track of current games and sessions. This will be a high activity table with many users and it makes much more sense to have this one extracted into (or left alone as) its own table, just like it makes sense to keep a user table and a session table separate.

Real world tip, Some people get obsessive about making a single table contain every thing. Thats a lot of hard work and I would not recommend you go that route. Sometimes having an occasional extra table when appropriate helps you stay sane.

The actual dragon data is really scoped to the 3 tables below. So we will disregard the current power table one for now, and see if we can consolidate these 3 into 1 single table (#win) and at the same time offer a way to do advance queries (#double\_win).

Here are the 3 tables we had:

###### DragonStats

| **dragon\_name (PK)** | **damage** | **description** | **protection** | **family** | **location** | **location\_city** | **location\_state** | **Location\_neighborhood** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cassidiuma | 7 | Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen's most loved and feared warrior. | 10 | red | usa | las vegas | nevada | spring valley |

We already have done that one :). So one down, two to go

It now looks more like this, with UUIDs the Primary Key and location consolidated (for advanced querys later).

###### single\_dragon\_table

| **PK (PK)** | **dragon\_name** | **damage** | **description** | **protection** | **family** | **location** |
| --- | --- | --- | --- | --- | --- | --- |
| c38af3d5-8c03-4beb-bcdb-5c4942cbc683 | Cassidiuma | 7 | Cassidiuma is the personal protector and knight of the dragon queen Methryl. She is the queen's most loved and feared warrior. | 10 | red | usa#kansas#colby#poplar st |

So what about the other two tables that currently look like this?

###### DragonBonusAttack

| **breath\_attack(PK)** | **description** | **extra\_damage** | **range (SK)** |
| --- | --- | --- | --- |
| acid | spews acid | 3 | 5 |
| electricty | bolts fly from mouth | 3 | 5 |
| water | high pressure jet over a large area | 1 | 10 |
| fear | Prevent all attacks next round | 0 | 4 |
| fire | Short blast of fire | 8 | 4 |

###### DragonFamily

| **breath\_attack** | **damage\_modifer** | **description** | **family (PK)** | **protection\_modifier** |
| --- | --- | --- | --- | --- |
| acid | -2 | Better defense | green | 2 |
| fire | 2 | Attacks faster | red | -2 |
| water | 1 | Happy in water | blue | -1 |
| fear | 0 | Prefers to bite | black | 0 |

The question is how do we bring these 2 into our dragon\_single\_table and create indexes that will allow the game developers (later on) to query the data in sane ways?

The first thing you should always do when working with DynamoDB Schema design is to **think about he queries first!**

So let's think of a few things that the site would maybe like to do in future iterations.

###### Here are some ideas:

1. Return all the dragons that can spew acid.
2. Return all dragons that are green.
3. Return the dragons in range attack (order).
4. Give me all the dragons that live in Arizona, USA (Dragons like the warmth apparently).

Let's address each of these one at a time and discuss what we would normally have had to do with say a traditional database or where we had multiple tables in DynamoDB.

## Part 2 - Step 1: Return all the dragons that can spew acid

If you had a traditional DB this would normally require looking up information in the DragonBonusAttack table first a bit like so:

SELECT breath\_attack FROM DragonBonusAttack WHERE description = "spews acid"

Then finding the family such as red (family).

SELECT family FROM DragonFamily WHERE breath\_attack = breath\_attack

Then finally:

SELECT \* FROM DragonStats WHERE family = family

Even in DynamoDB where we have multiple tables that would require a very similar approach.

Using the single table approach we can do better than that.

All we need to do is add items using the same primary key as follows:

| **PK (PK)** | **SK (SK)** | **Location** | **<attributes>** |
| --- | --- | --- | --- |
| <SAME-UUID> | stats | <Location\_country\_value>: <Location\_state\_value><Location\_city\_value>:<Location\_neighborhood\_value> | dragon\_name:<value>, protection:<value>, damage:, description:<value>, family:<value> |
| <SAME-UUID> | bonus |  | extra\_damage:<value>, range:<value>, bonus\_description: <value> |
| <SAME-UUID> | family |  | breath\_attack:<value>, damage\_modifier:<value>, protection\_modifier:<value>, family\_description: <value> |

The first time you seee the single table pattern it's mindbending I know, but bear with us..

Notice we had to change the attribute of description to family\_description to not conflict with the description attribute used for dragon stats, and we also did this for the bonus table's description for the same reason.

Real World Tip: We could use description as an overloaded attribute, which is very useful for reducing the amount of GSIs needed to do certain queries, refer to the course notes.

Notice that we are adding a Primary Key with the same value representing one dragon.

This might look very strange at first, but this is correct for what we are trying to do.

Normally you can't have a duplicate primary key, unless you have a Sort Key.

So as you can see above, we now have a Sort Key called SK and are assigning three string values to each of the 3 items. One called stats, one called bonus and one called family. These are strings of the words, and not variables. i.e the strings **"family"**, "**bonus**", and "**stats**".

The old family table had a family value, which we removed, because it is already there in the stats item for this dragon. We are also taking away breath\_attack out of bonus as it is already there in the family item.

We could have just duplicated it but it seems unnecessary, as none of our queries that we are envisioning would need it. **Hence the value of thinking about queries first**. It makes it a little less flexible in case you want to do ad-hoc queries. However that is the compromise you make by using this approach.

So the prior queries still work as expected, we haven't messed anything up there ;). All we have really done here is added a couple of new items with the same UUID assigning a different SK to each one and removed some duplication.

So before we go ahead and create an improved single table with this new structure, it is always better to measure twice and cut once.

So let's really think about our queries some more and figure out if we are going to need any GSIs.

It is going to be easier to do all that, in one go.

So query number 1) We want to **Return all the dragons that can spew acid**.

Let's think about that.

This needs to be done in two stages.

Stage 1) Get all the UUID of all the dragons that can spew acid.

This will require an index which we could call bonus\_description\_index with bonus\_description as the Primary Key.

Stage 2) provide the UUIDs to a new scan that can return the dragon details (stats).

This will require exactly the same index as we created before called dragon\_stats\_index. So we will definitely keep that as a GSI moving forward.

Even without coding any of this, we can see that we have a sensible enough schema and GSI to accommodate both of these actions.

So we can move on. We will do the code for this stuff in a bit, but for now, let's continue conceptualizing our other queries to make sure that **a)** they are all possible and **b)** if we are going to need some more GSIs .

## Part 2 - Step 2: Return all dragons that are green.

Let's see if this is possible using what we already have proposed, or if we need to modify our proposed table again.

It turns out that the family e.g green is easily accessible in our single table on the item that has the Sort Key of stats. Remember we removed it from the other items as it was not needed.

If would be as simple as this

aws dynamodb scan --index-name dragon\_stats\_index --table-name single\_dragon\_tables --filter family = "green"

Ok, I like that one, it's simple and it will work using an existing index with a scan and filter.

Real World Tip: Some people get hung up on only using queries and never using scans, however think about this one: We have 48 cards, so 12 of them will be any color, we are essentially scanning and throwing away 36 cards from the scan, this is not something you should concern yourself with. For large scale or high volume apps where scans can become kind of unwieldy or expensive, sure, optimize further, create GSI, but for this one (and many real world situations) *scan and filter* is just fine. #evils\_of\_early\_optimization

That's 2 out of 4 conceptualized, let's keep going.

## Part 2 - Step 3: Return the dragons in range attack order

Hmm, this one is a little more interesting as to brings something back in order (or a reverse order). For this we really need to be using number attributes on the Sort Key.

The problem is our Sort key is taken, Doh however we can create a GSI with range as the Sort Key

###### Proposed range\_index

| **SK(PK)** | **range (sk)** | **<attributes>** |
| --- | --- | --- |
| bonus | 5 | PK : <UUID> |

Note that the PK is the old SK. This is a reverse index, and super useful for look-ups in a single table.

Although you could just scan the table, that won't give you the range order. You need to be able to specify what we call the ScanIndexForward key to true (default - ascending) or false (defending).

As we want descending, we need to specific ScanIndexForward as false and you need to use a query for that, as it is not an option for a scan. Which means proving a Primary key, which is luckily the value "bonus".

I hope you are starting to see the value of single tables already.

So query 3 is super easy, we just need a simple range index.

## Part 2 - Step 4: Give me all the dragons that live in Arizona, USA

And finally (thank goodness I hear you say) for query number **4 - Give me all the dragons that live in Arizona, USA**

We recently removed all the location info, so you can't do the following anymore

aws dynamodb scan --index-name dragon\_stats\_index --table-name single\_dragon\_tables --filter location\_state = "arizona"

This would be a horribly inefficient scan anyway, as there are many dragons not in Arizona.

You might be thinking that you could have set up a query on location\_state..had we not removed it ;)

However good news!

Remember earlier we created a **composite** attribute like this.

usa#arizona#flagstaff#lake mary rd

All we need to do is create a **reverse GSI** called location\_index with SK as the Primary Key, and just like you did with the stats index, project the dragon info that you need returning, like the following:

###### Proposed location\_index

| **SK (Pk)** | **Location (SK)** |  |
| --- | --- | --- |
| stats | <Location\_country\_value>:<Location\_city\_value>:<Location\_neighborhood\_value> | dragon\_name:, protection:, damage:, description:, family: [uuid-PK added by default] |

Note you made the Location the Sort Key, and this is KEY excuse the pun!

It is critical that the new Sort Key is present so we can leverage the features of DynamoDB that allow us to place **functional conditions** in our queries.

Look how cool this query is using the single table concept.

Super efficient!

aws dynamodb query --index-name location\_index --table-name single\_dragon\_tables --keycondition location = starts\_with("usa#arizona")

Not powerful enough? How about give me every dragon spotted (near me) in Spring Valley in Las Vegas?

aws dynamodb query --index-name location\_index --table-name single\_dragon\_tables --keycondition location = starts\_with("usa#nevada#las vegas#spring valley")

#win

**Wow**, you got through all of the concepts, now it's time to code it!

Before you go to Marry with the new index5.html to show her the base functionality can be easily replicated and simplified using singe table pattern, you really want to be able to show here these queries and not just give her an explanation of how it would work.

###### In this final part of this lab (and the course actually) you will do the following:

1. Create a **new** single table (singe\_table\_improved) based on the schema we just discussed, ensuring we have all the GSIs we need for all our potential queries.
2. Seed it correctly with data, similar to before, just accommodating the new schema.
3. Create a few .js files that you can use to show example queries.
4. Hand everything over to her devs and let them take it from here, so you can get your chocolate cake off Mary for all your hard work .

## Part 2 - Step 5: Create a new single improved table

1. Open create\_improved\_single\_table\_and\_all\_indexes.js
2. Edit the s like normal.
3. Run it
4. node create\_improved\_single\_table\_and\_all\_indexes.js

You should see:

null { TableDescription:

{ AttributeDefinitions: [ [Object], [Object], [Object], [Object], [Object], [Object] ],

TableName: 'improved\_single\_dragon\_table',

KeySchema: [ [Object], [Object] ],

TableStatus: 'CREATING',

CreationDateTime: 2019-06-05T17:03:57.138Z,

ProvisionedThroughput:

{ NumberOfDecreasesToday: 0,

ReadCapacityUnits: 0,

WriteCapacityUnits: 0 },

TableSizeBytes: 0,

ItemCount: 0,

TableArn: 'arn:aws:dynamodb:us-east-1:000000000000:table/improved\_single\_dragon\_table',

TableId: 'c28b30ad-8ef9-4b95-a1e5-e83a5299fc81',

BillingModeSummary: { BillingMode: 'PAY\_PER\_REQUEST' },

GlobalSecondaryIndexes: [ [Object], [Object], [Object], [Object] ] } }

⚠️ This may take some time, do not move on until that table is ACTIVE, or part 2 step 6 will fail. You might want to grab some coffee, it's annoyingly slow.

## Part 2 - Step 6: Seed it correctly with data

1. Open seed\_improved\_table.js
2. Edit the s like normal.
3. Run it:
4. node seed\_improved\_table.js #just once please

You should see this:

seeded

## Part 2 - Step 7: Create a few .js files that you can use to show example queries.

1. Open query\_1.js (Return dragons that can [spew acid] ).
2. Edit the s like normal.
3. Run it:
4. node query\_1.js test "spews acid"

You should see this. We just projected the dragon name for brevity.

24 dragons found that spews acid

null [ { dragon\_name: { S: 'Nightingale' } },

{ dragon\_name: { S: 'Castral' } },

{ dragon\_name: { S: 'Bahamethut' } },

{ dragon\_name: { S: 'Magnum' } },

{ dragon\_name: { S: 'Ragnorl' } },

{ dragon\_name: { S: 'Pradumo' } },

{ dragon\_name: { S: 'Sheblonguh' } },

{ dragon\_name: { S: 'Blackhole' } },

{ dragon\_name: { S: 'Midnight' } },

{ dragon\_name: { S: 'Dexler' } },

{ dragon\_name: { S: 'Shadow' } },

{ dragon\_name: { S: 'Sonic' } },

{ dragon\_name: { S: 'Cassidiuma' } },

{ dragon\_name: { S: 'Mino' } },

{ dragon\_name: { S: 'Smolder' } },

{ dragon\_name: { S: 'Amaron' } },

{ dragon\_name: { S: 'Samurilio' } },

{ dragon\_name: { S: 'Prythus' } },

{ dragon\_name: { S: 'Shulmi' } },

{ dragon\_name: { S: 'Warcumer' } },

{ dragon\_name: { S: 'Tornado' } },

{ dragon\_name: { S: 'Lucian' } },

{ dragon\_name: { S: 'Havarth' } },

{ dragon\_name: { S: 'Jerichombur' } } ]

1. Open query\_2.js (Return all dragons that are [green]).
2. Edit the s like normal.
3. Run it.
4. node query\_2.js test "green"

You should see this: We just projected the dragon name for brevity.

null [ { dragon\_name: { S: 'Ragnorl' } },

{ dragon\_name: { S: 'Pradumo' } },

{ dragon\_name: { S: 'Cassidiuma' } },

{ dragon\_name: { S: 'Amaron' } },

{ dragon\_name: { S: 'Samurilio' } },

{ dragon\_name: { S: 'Prythus' } },

{ dragon\_name: { S: 'Shulmi' } },

{ dragon\_name: { S: 'Warcumer' } },

{ dragon\_name: { S: 'Havarth' } },

{ dragon\_name: { S: 'Jerichombur' } } ]

1. Open query\_3.js (Return the dragons in range attack order - highest first).
2. Edit the s like normal.
3. Run it.
4. node query\_3.js test

You should see this: (note we added *range* to each item sent back).

....more items...

....

dragon\_name: { S: 'Isilier' } },

{ pk: { S: '2bf1f72f-90b9-4695-a9ad-c582dfae252e' },

range: { N: '4' },

damage: { N: '7' },

protection: { N: '9' },

family: { S: 'red' },

description:

{ S: 'Ruby has a skin and coat that\'s as hard as gems. This gives her extra defense against her enemies.' },

dragon\_name: { S: 'Ruby' } } ]

Dragons in range order:

1. Open query\_4.js (Give me all the dragons that live in Arizona, USA).
2. Edit the s like normal.
3. Run it.
4. node query\_4.js test "usa#arizona"

You should see this:

null [ { location: { S: 'usa#arizona#chandler#w german rd' },

damage: { N: '7' },

sk: { S: 'stats' },

family: { S: 'green' },

description:

{ S: 'Ragnorl is a rogue dragon, disowned from his own tribe. He can change colors to blend with the earth around him.' },

pk: { S: '04e39412-ed8e-41ea-b7f8-e5d6ff73c002' },

protection: { N: '7' },

dragon\_name: { S: 'Ragnorl' } },

{ location: { S: 'usa#arizona#flagstaff#lake mary rd' },

damage: { N: '9' },

sk: { S: 'stats' },

family: { S: 'black' },

description:

{ S: 'Sonic has black spikes that can penetrate his enemies. He has a spiked tail that can attack his opponents.' },

pk: { S: 'dbe13c27-2e85-4480-aae8-e003e00cad31' },

protection: { N: '8' },

dragon\_name: { S: 'Sonic' } },

{ location: { S: 'usa#arizona#mesa#e adobe st' },

damage: { N: '6' },

sk: { S: 'stats' },

family: { S: 'blue' },

description:

{ S: 'Frealu has an ice breath that can freeze her enemies into a paralyzed state. She is from the souther water tribe.' },

pk: { S: 'e21bb786-d240-416a-ac12-7f65fced089b' },

protection: { N: '6' },

dragon\_name: { S: 'Frealu' } },

{ location: { S: 'usa#arizona#page#morgan rd' },

damage: { N: '8' },

sk: { S: 'stats' },

family: { S: 'blue' },

description: { S: 'Always hiding in shadows' },

pk: { S: '3c1ce221-6df1-417c-bb17-8b2172d37407' },

protection: { N: '6' },

dragon\_name: { S: 'Sanguia' } },

{ location: { S: 'usa#arizona#tempe#e laguna dr' },

damage: { N: '4' },

sk: { S: 'stats' },

family: { S: 'red' },

description:

{ S: 'Firestorm can summon a fire storm of hail and rain, that burns his opponents.' },

pk: { S: 'a60085d4-84dd-4859-9bf2-2b27c3519751' },

protection: { N: '9' },

dragon\_name: { S: 'Firestorm' } } ]

**Congrats** you can now show Mary these new queries from your laptop, and convince her about the benefits of using the single table pattern.

You're' done!

Oh wait, there's one more **critical** task to complete…Text Mary and get her to bring over that chocolate cake!

We hope you had lots of fun working with DynamoDB in a web application setting, and learned a few new things too.

Congrats on sticking with it, there was a lot in here!

….See you next time