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 he 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 be notified 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 *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 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.
3. Ensure you are in the root directory, and close any 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 AWS 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-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).
6. To unzip the lab4.zip file, run the following command:

   ```
   cd ~/environment
   unzip lab4.zip
   ```

   In your Cloud9 filesystem, you should see a lab4 folder.
7. To keep things clean, run the following commands to remove the zip:

   ```
   rm lab4.zip
   cd lab4
   echo "done"
   ```

8. 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 X-Ray at the API Gateway 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. Choose the **Resources** link on the left menu. Then, click **Actions > Deploy API**) (prod).

Now that 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, but recommended)**

Step 2: Explore the CloudWatch Metrics of DynamoDB (optional)

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

different metrics.

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 'FMIs', no scripting.

Just for a change!

I think it's valuable for everyone to learn the basics of the CLI. You don't have 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 AWS Cloud9 environment). This will make the commands below easier to work with.

Step 3: Get alerted on UserErrors

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

1. In the **bash terminal** of your Cloud9 environment, 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-1:123456789012: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-1:123456789012: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.

2. 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"
}
```

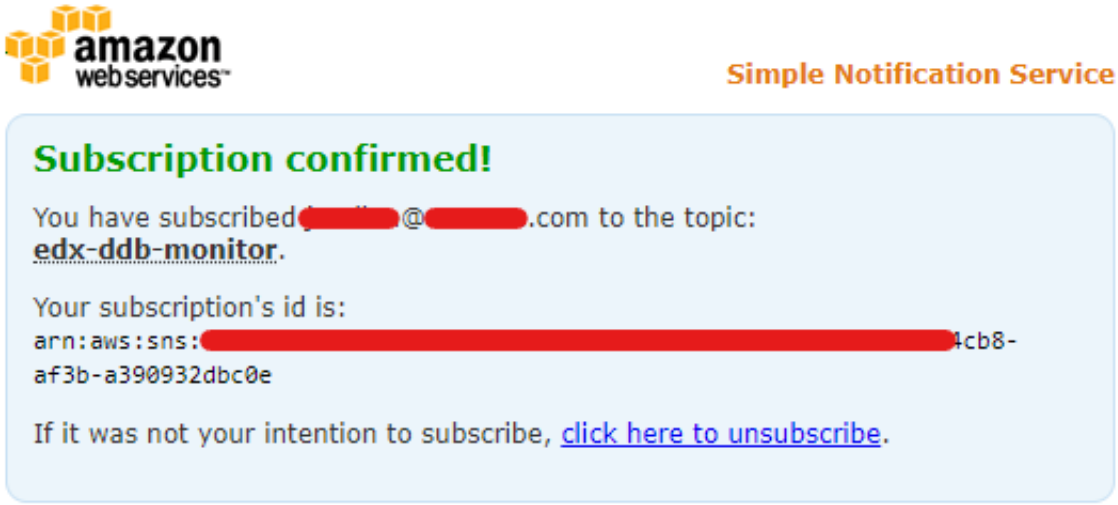
3. You will receive an email at the address that you provided. It will be from **no-reply@sns.amazonaws.com** and will contain a link to confirm your subscription. In that email, click on the link **Confirm subscription**. The email will be similar to the following:

```
You have chosen to subscribe to the topic:
arn:aws:sns:us-east-1:123456789012:edx-ddb-monitor

To confirm this subscription, click or visit the link below (If this was in
error no action is necessary):
Confirm subscription

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:



4. To create the Amazon CloudWatch Alarm, execute the following command in the **bash terminal**. 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.

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

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

You hate yourself for pushing untested code and overwriting your old "tested" code, but you brace yourself and consider that the learning experience is fixing anything that happens to go wrong is worth it. Besides, it's only you and Mary's team that are seeing this thing 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).* Once uploaded, you should hit the website a few times and do a few searches on it to generate some metrics and trigger the CloudWatch and X-Ray monitoring you just set up.

1. To overwrite your tested code to what Steve created, execute the following CLI command from the **bash terminal** in **Cloud9**.

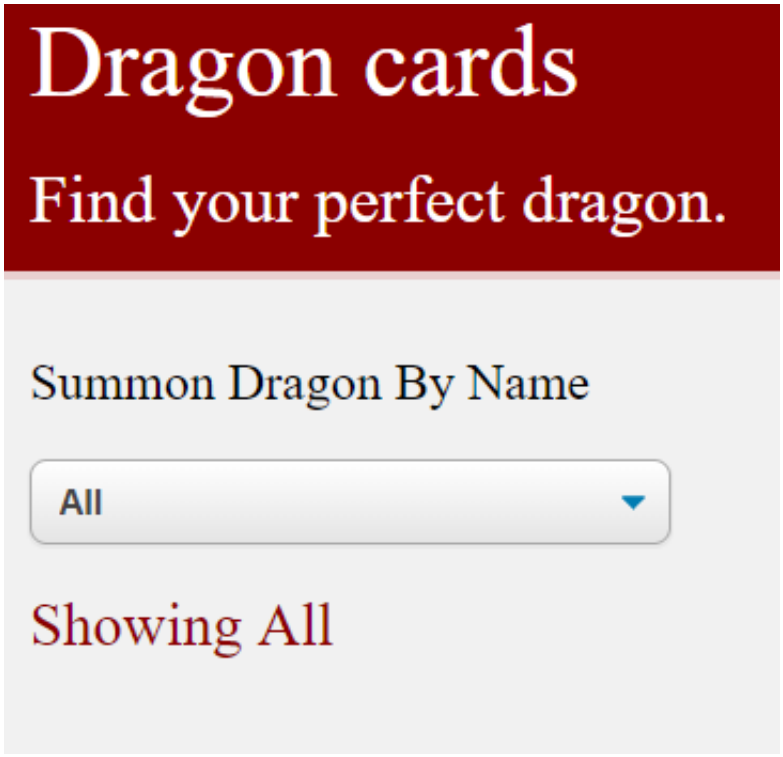
```
cd ~/environment/lab4
aws lambda update-function-code --function-name DragonSearch \
--zip-file fileb://resources/steve-code.zip
```

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. From **Cloud9**, click the **AWS Cloud9** button next to File.
2. Select **Go To Your Dashboard**.
3. Click the **Services** menu and choose **S3**.
4. Click on the S3 bucket you created in Exercise 2.
5. Click on the **index2.html** object.
6. Click on the link of the **Object URL**.

   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 dragons - you have seen them before!

7. Press refresh a few times
8. Now do a search for `Fireball`
9. Then do a search for `Dexlar`.
10. Then select `All`

..YEP `All` is broken.

You remember that the 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!

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)

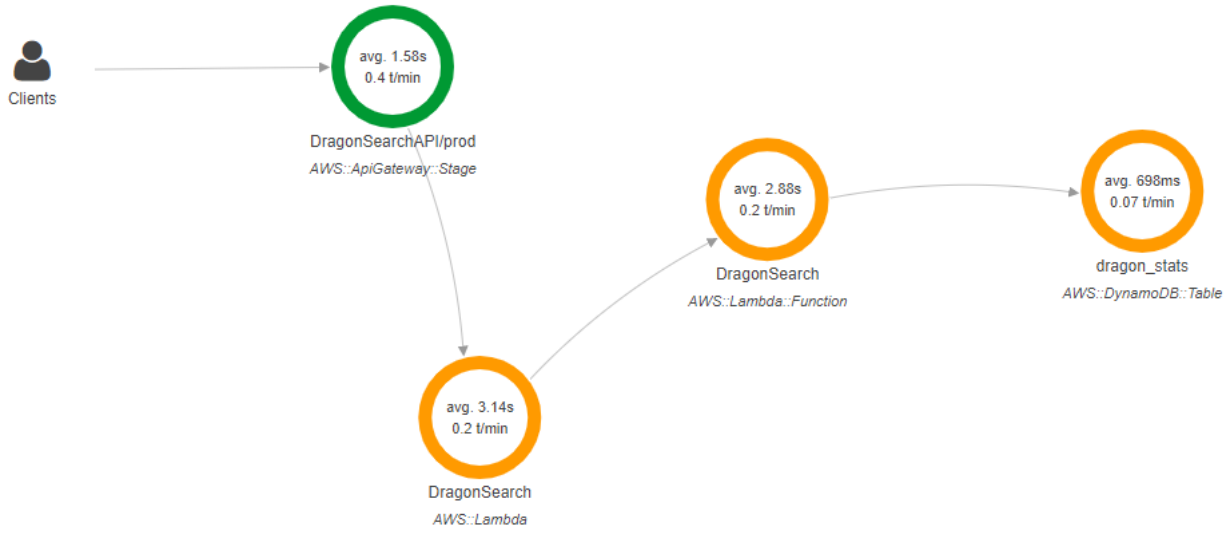
You have just received an alarm via an email that a UserErrors on a DynamoDB table just happened. As seen in the documentation, a UserErrors means that a request to DynamoDB generated a 400 status code. This means that there is a problem with the request sent. You know the error is coming from the Lambda 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 will help you debug the issue, and allow you to gain data on API Gateway, as well as information on what's happening with DynamoDB. Feel free to look at Steve's code to see how simple it was to implement X-Ray instrumentation.

Let's start troubleshooting by looking at X-Ray.

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. 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.



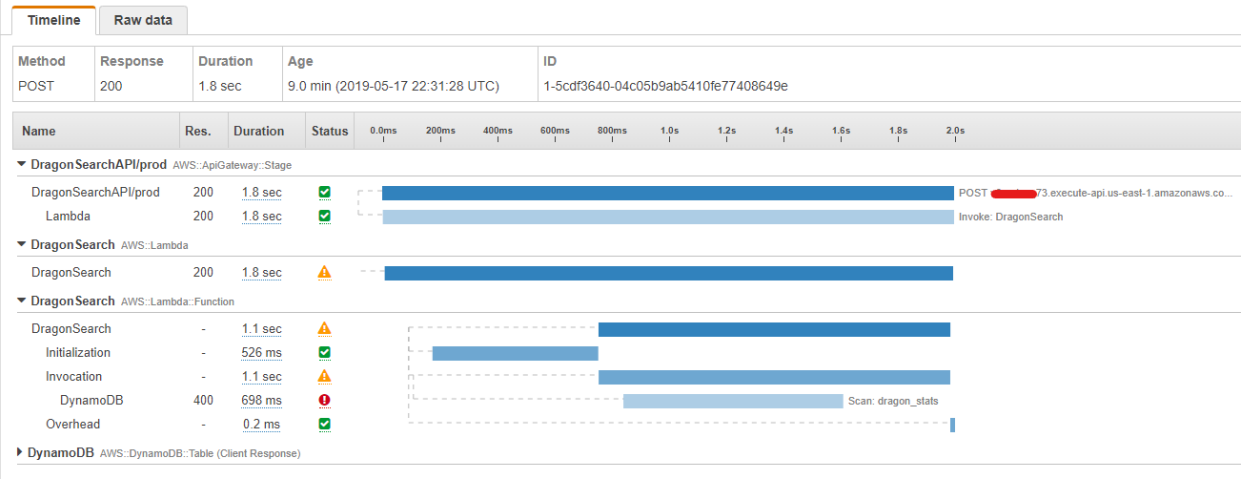
   As you can see the client (Mary/you) sent a request to *DragonSearchAPI* in the *prod* stage. That request was successful as it's represented in green. You can click on the Map legend link on the right to know what the colors mean. Yellow means there was a 4xx error which we expect here. It was then sent to AWS Lambda the service which then sent the request to your *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 need to fix his code.

5. Click on the **dragon_stats** in the map which will open a menu on the right.
6. Click on the **View traces >** button.
7. 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.

8. 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.

9. Expand **DynamoDB** by clicking on the breadcrumb next to it.
10. Click on **DragonSearch** under the DynamoDB section.
11. 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.

12. Click the **Close** button.
13. 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.

14. 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.
4. Click on the **Monitoring** tab.
5. Click on the button **View logs in CloudWatch**.
6. 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.
7. 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.
8. 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. logs, 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 are causing the issue.

The reserved keyword **family** is used for one of the attribute 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 using the `justOneDragon` test.

It should be a success, as this uses query and not 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` which uses 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, find the **scanTable** function.
2. To understand how reserved keywords, ExpressionAttributeNames and ProjectionExpression are all related, visit this documentation.

3. Now that you understand the link, open the node AWS SDK for the scan operation and look at how you can modify the **params** variable to make the code work.
4. You can directly modify the code in the Lambda console, **however** let's use our *usual process*. Let's test it in AWS Cloud9 first, then move it to Lambda.
5. Close the Lambda console un-edited and head back to AWS Cloud9.
6. Open `lab4/steve_code.js` and fix the `params` variable and the `ProjectionExpression` variable in the `ScanTable` function. *Note: this is all the coding you need to do in this lab.*
    - 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.
7. Before you test the code check your proposed code with the solution file.
8. Note Steve's conditional installation of X-Ray 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 **AWS Cloud9 terminal.**
2. `node steve_code.js test 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 test 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' } } ]
```

Now it is all working! Copy this code to your clipboard and paste it into the Lambda function `DragonSearch`, overwriting the existing code with your new tested code.

Don't forget to **Save** the new code in the Lambda console, and use the `DragonScan` test case to trigger the scan.

In the test results, you should see all the dragons returned.

Congrats! You fixed his code and published it.

Step 7: Learn how to do pagination in DynamoDB and to instrument the code with X-Ray (optional)

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.

Pagination (*optional step*, 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.

X-Ray (*optional step*, 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 returned every dragons. 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.

As you will test the function directly from the console, you will need to enable tracing in your Lambda function directly. Before, that was enabled via API Gateway, but you will be bypassing it for this test.

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 8.1): Latency of a Scan with Filter

1. Click the **Services** menu and choose **Lambda**.
2. Click on the Lambda function **DragonSearch**.
3. Scroll down to the **AWS X-Ray** card.
4. Click the check mark next to **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. Scroll down to the **AWS X-Ray** card once again.
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, find the **justThisDragon** function and spend time understanding how it works.

   This function creates a *params* object and uses the *dragon_name_str* sent as a parameter to build a *FilterExpression*. It then sends a *scan* to DynamoDB. When the data is returned, it sends back what it found using the *cb* variable containing the callback of Lambda.

   The first parameter of the callback is the error. If there are no errors, then *null* must be passed as the first parameter and the data to return as the second parameter.
4. Now that you understand the code, open the node AWS SDK for the query operation and look at how you could transform this scan into a query.
5. 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.
6. 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:


```
[
    {
        "location_neighborhood": {
            "S": "spring valley"
        },
        "damage": {
            "N": "9"
        },
        "location_city": {
            "S": "las vegas"
        },
        "family": {
            "S": "green"
        },
        "description": {
            "S": "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat."
        },
        "protection": {
            "N": "3"
        },
```

```
        "location_country": {
            "S": "usa"
        },
        "location_state": {
            "S": "nevada"
        },
        "dragon_name": {
            "S": "Cassidiuma"
        }
    }
]
```

Step 8.3): Latency of a Query

1. Scroll down to the **AWS X-Ray** card on the configuration tab of the **DragonSearch** Lambda function.
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 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 AWS Cloud9 IDE.

5. Open `scan_dragon_optimized_projection.js` and replace the FMIs in order to project only the attributes you need for the website:
    - dragon_name
    - family
    - protection
    - damage
    - description

Once you have improved the code, save the file.

- If you have issues, you can find the solution of the code in the file `scan_dragon-optimized-projection.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.

To test your code, 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 the code from `scan_dragon_optimized_projection.js` to your clipboard and overwrite the function code in the `DragonSearch` Lambda function.
2. To test the code in the Lambda console, 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:

```
[
    {
        "family": {
            "S": "green"
        },
        "damage": {
            "N": "9"
        },
        "description": {
```

```
            "S": "Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
commodo consequat."
        },
        "protection": {
            "N": "3"
        },
        "dragon_name": {
            "S": "Cassidiuma"
        }
    }
]
```

Scroll down to the **AWS X-Ray** card in the configuration tab of the **DragonSearch** Lambda function console and click on **View traces in X-Ray**. Then, 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.

Awesome! 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.