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 AWS 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
```

2. You will need get the files that will be used for this exercise, 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-members/lab5.zip \
-P /home/ec2-user/environment
```

3. 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).
4. To unzip the workdir.zip file, run the following command:

```
unzip lab5.zip
```

This may take a few moments to unzip.

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

```
rm lab5.zip && cd lab5
```

6. 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 going 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 should be in your AWS Cloud9 environment.

You want to be able to let a user log in with their `email_address` so you will need to create a Global Secondary Index, using `email_address` as the Primary Key.

User table

user_name	first_name	email_address (PK)	password
dave937434	Dave	dave3423@dragonedx.com	sjdhjsgdjsgdjhsgd34erdtfgd

Email Index (no need to project first name)

user_name	email_address(PK)	password
dave937434	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 `users` and 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).

	Name	Status	Type	Partition key	Sort key	Attributes	Read capacity	Write capacity
○	email_index	Active	GSI	email_address (Strin	-	user_name, email_address, password	On-Demand	On-Demand

*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 and out, and this needs to have a session table to maintain your users logged in status.

Later in this lab, in 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 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. 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**.
3. Save the file
4. Go to the terminal and run your file using the respective run command below

```
node create_sessions_table.js
```

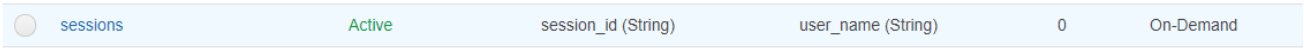
IF YOU GET STUCK, OR IT IS NOT WORKING, SIMPLY COPY THE CODE SITTING IN THE RESPECTIVE SOLUTION FOLDER.

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.



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 he 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

Note as there can often be a delay in purging expired items (up to 48 hours) using this method, we are going to need 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. Later on, we will set the TTL for 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

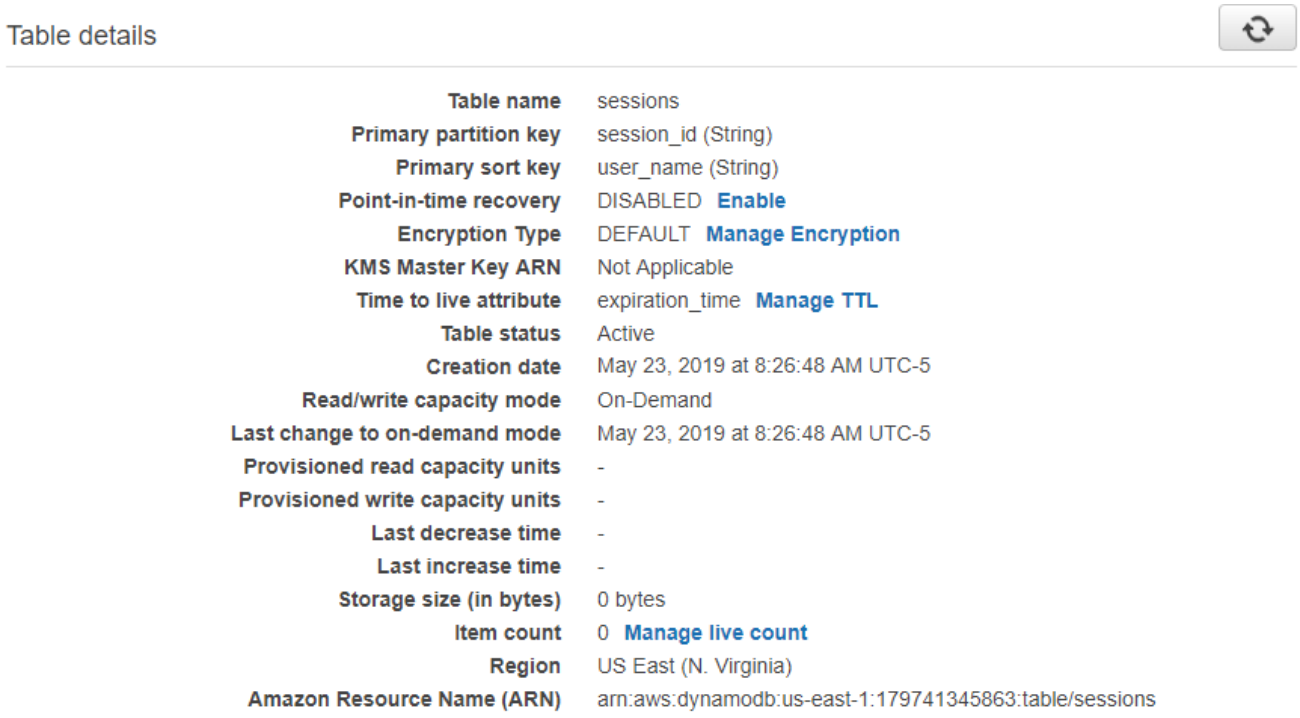
```
node enable_ttl.js
```

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'
} }
```



Step 4: Upload user data from a JSON file.

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

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 plaintext passwords. 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. We should not be storing passwords in clear text in the database.

You will notice a line in the code that hashes the temporary plaintext password. It uses 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 and run this following line.

```
cd /home/ec2-user/environment/lab5
```

Then run:

```
npm install bcrypt
```

(Ignore all warning)

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 user the the user table that you creared earlier.

1. Open up the `upload_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

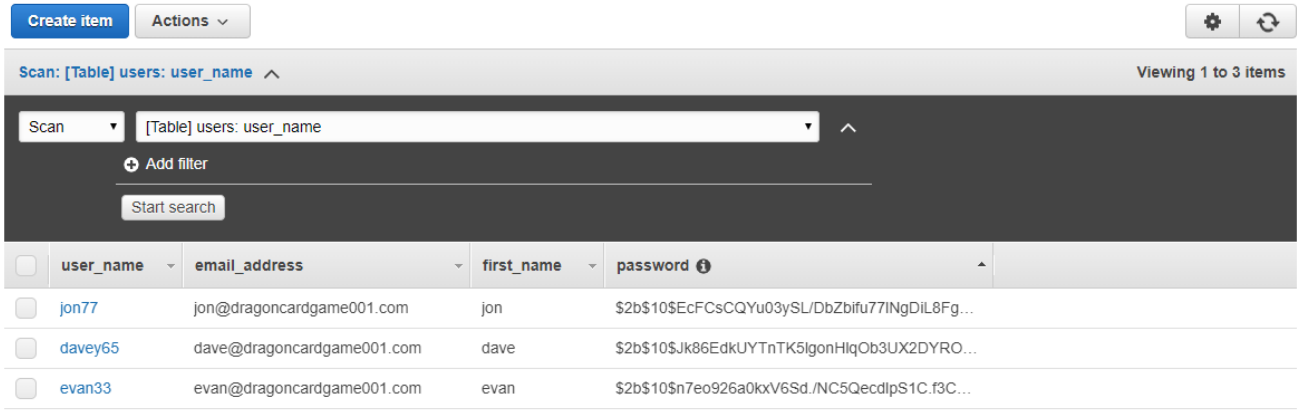
    ```
    node upload_and_hash_passwords.js
    ```

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. Second, 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).

`cd /home/ec2-user/environment/lab5`

Then run this installation command to bring in our random string generator (`for our session_id_str` 's).

`npm install uuid`

3. Have the SDK docs open (as above) to help you
    1. Replace the <FMI> sections of the code in that file.
4. Save the file
5. Go to the terminal and run your file using the respective run command below.

First try 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 (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, you would be able to refresh the dynamo 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 we need to do is 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).

- For now, let's switch the session to 20 minutes instead of one minute. Go back to the code (`login.js`) and swap the **SESSION_TIMEOUT_IN_MINUTES_INT = 1**; on line 34 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.

As you have already created a Lambda function before, the concepts should be pretty familiar to you. However, 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)

```
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.

```
cd /home/ec2-user/environment/lab5
```

2. Install the packages into a folder called `node_modules`

```
npm install --prefix /home/ec2-user/environment/lab5 uuid bcrypt
```

3. Ignore any warnings.
4. Now zip it all up

```
zip -r login.zip node_modules login.js
```

You will see that a **login.zip** file has been created in the following path `/home/ec2-user/environment/lab5/` in AWS 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. 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**.
3. Before saving the file, change the Lambda Function **Runtime** on **line 39** of the `createLambdaFunction` function in the `create_and_publish_login.js` file in from its existing value of `nodejs8.10` to a **new** value `nodejs10.x`.
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 AWS Cloud9 terminal).

```
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:

```
{
    "email_address_str": "dave@dragoncardgame001.com",
    "attempted_password_str": "spaceship"
}
```

7. Choose **Create**.
8. Choose **Test**.
9. You should see it fail with (expand details):

```
{
  "errorMessage": "password does not match email"
}
```

10. Now use the correct password. Create a new test under **Configure test events** called **realPassword** and paste the following contents.

```
{
    "email_address_str": "dave@dragoncardgame001.com",
    "attempted_password_str": "apple"
}
```

11. Run the test and you should see something like this:

```
{
  "user_name_str": "davey65",
  "session_id_str": "9f15f3ef-4700-43e7-b602-f11f8b892727"
}
```

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 everything unchecked)
6. With `/login` highlighted, Choose **Actions** and **Create Method**.
7. Choose **POST** and click the checkmark icon next to it.
8. Leave the **Integration type** as **Lambda Function**.
9. Make sure Lambda proxy is unchecked.
10. In the **Lambda Function** type in **LoginEdxDragonGame**.
11. Uncheck **Use Default Timeout** and choose a custom timeout of `10000` (10 seconds)
12. Choose **Save**.
13. Choose **OK** at the add permissions pop-up.
14. 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"
}
```

14. Select the `/login` under **Resources** and choose **Actions** and **Enable CORS**.
15. Check **DEFAULT 4XX** and **DEFAULT 5XX** (ignore any warnings).
16. Choose **Enable CORS and replace existing CORS headers**.
17. Choose **Yes, replace existing values**.
18. Choose **Actions** and **Deploy API**.
19. Choose **prod** and choose **Deploy**.

Step 6C): Test the website.

1. Go to your website but use `/index3.html` instead of `/index2.html`. You should be presented with a login screen.
2. Try and login to the website with a fake password:

```
dave@dragoncardgame001.com
yellow
```

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

3. Now try using the correct password:

    ```
    dave@dragoncardgame001.com
    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, bypassing the website login protection.

For example try this in the AWS Cloud9 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:

```
{
  "damage": 4,
  "description": "Nightingale uses her song to lull and seduce her opponents. She
is deadly.",
  "dragon_name": "Nightingale",
  "family": "black",
  "location_city": "eagan",
  "location_country": "usa",
  "location_neighborhood": "short line",
  "location_state": "minnesota",
  "protection": 6
}
```

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

We need to update our `DragonSearch` lambda function so that when a request comes in to it, it will reject anything without a valid username and 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. We could edit the file in AWS Cloud9 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 and test it there.

1. Navigate back to the Lambda console choosing **Services** and searching for **Lambda**.
2. Choose the **DragonSearch** function. *Don't worry, we're not going to make you folks code this step.*
3. Replace the contents of **index.js** with the contents from `/lab5/resources/protected_dragon_search.js`.
4. All this code does is check for a valid session before moving on.
5. Choose **Save**.
6. Run one of the old test cases you created earlier called `justOneDragon`
7. You should get the following error message, which means it is now protected.

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

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 <FMI>)

```
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!