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).

4. To unzip the `lab7.zip` file, by running the following command:

```
cd ~/environment
unzip lab7.zip
```

This may take a few moments in your AWS Cloud9 filesystem.

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

```
rm lab7.zip && cd lab7
```

6. 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 AWS Cloud9 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.
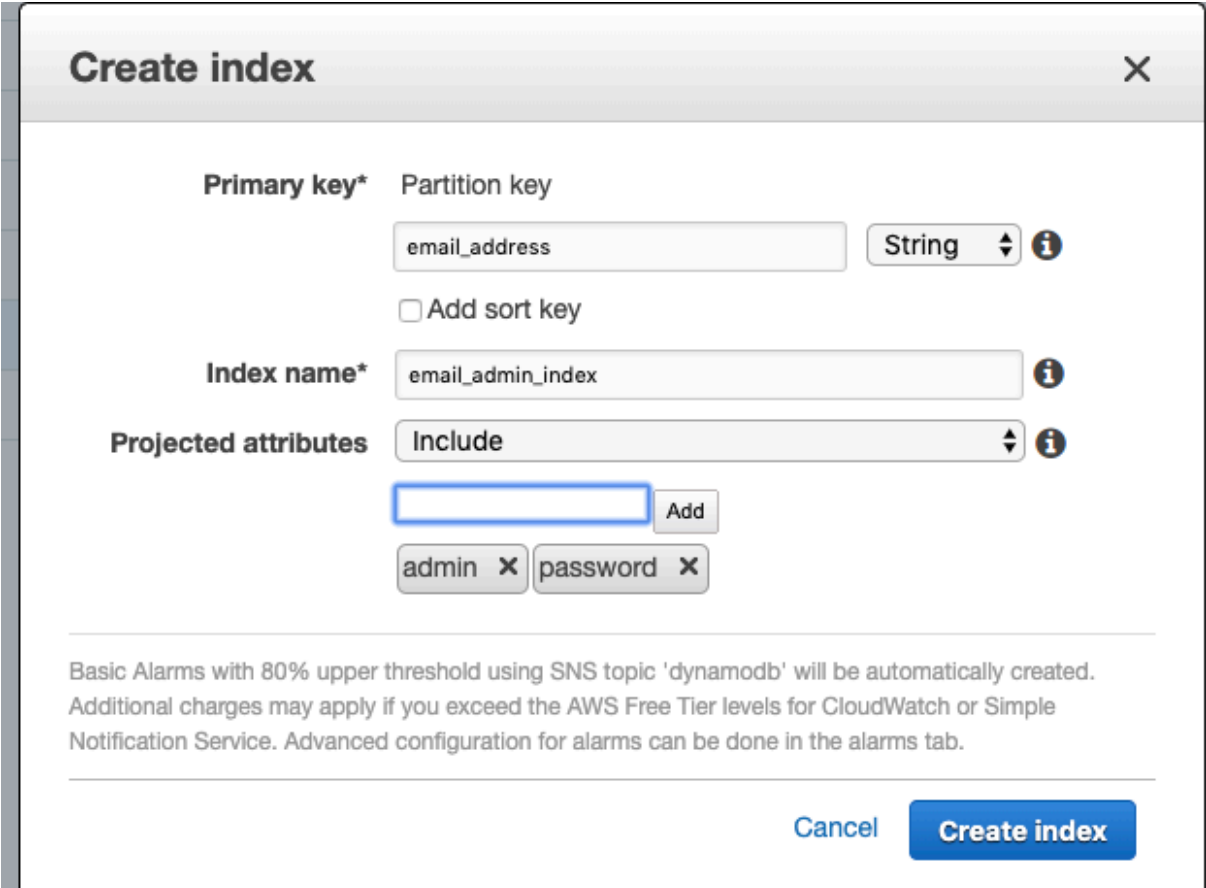
One challenge here is that you created a searchable index on the table that is being used during logins.

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). Do 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 the `admin_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 this: (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.

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

   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.html` and log in as Dave if you are not logged in already

   ```
   dave@dragoncardgame001.com and apple
   ```

You will not see any edit functionality (as Dave is not an admin).

2. Logout (press `logout davey65`) and log back in as Mary.

   ```
   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 <u>updating existing items</u>. 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 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, because it will mess up the card counts, as we have a certain amount 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.

3. Once you have coded `edit_1.js`, save the file.

4. <u>Before</u> 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 and you can do this using this command (just like before).

```
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 lab7, called `update_to_frost.json`

If you open `update_to_frost` you will see it 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 `Atlas` and 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:

```json
{
    "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 <u>fails</u> 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, we will finally delete the old record.

It may seem 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 and 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 - 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 (don't do this yet) 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 in use already.

Step 6: Conditions

We need to find a way to ensure that updates that attempt to create a negative protection value 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 "conditions" 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.

   ```
   node updated_login.js test mary@dragoncardgame001.com pears
   ```

   Then use the session id to make your calls

   In `update_to_fireball.json`, the code attempts to change `Fireball`'s name to `Blackhole`. Remember that we have a dragon already called `Blackhole`, so this is a good test case to check our code with.

   Your code **should** look like the following test case. If it doesn't, update it to match. Remember, 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

5. Try this and hope it **fails** ;)

```
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 `-8` the 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 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.

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.

   ```
   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
}
```

This should FAIL, because we already have a dragon 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, let's try this one:

Use `update_to_fireball.json` and ensure the code looks like this:

```
{
    "description_str": "Always hiding in shadows",
    "dragon_name_str": "Sanguia"
}
```

You should get the following result:

```
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 your 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. Check the **Active Tracing** checkbox under the **AWS X-Ray** card.
12. Choose **Save**.

Now head over to API Gateway, as you have already tested the code previously.

1. Choose **Services** and search for **API Gateway**.
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**
13. 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 (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-coding it. This would allow for updates and name changes to refresh in the list.

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 🤔