Exercise 3.2 Amazon DynamoDB

Exercise 3.2: Creating a DynamoDB table

We have our Lambda function all wired up, so all we need to do is change the function code to query a database.

Which we don't have.

Yet ;)

We are going to create a DyanmoDB table, and use lambda to seed it with data that we will provide in CSV format. Then test query the database, and run a few test cases for Lambda to make sure it is all wired up correctly.

As far as the dynamo table goes we are not going to create a sort key or an index. We are also foregoing a clever schema, because this lab is a one trick pony that only needs to do one simple search: "What is the temp of this city".

Pretty easy structure, its almost spreadsheet like!

Out Table will look a bit like this

Primary Key [SC]	**T**
NORTH LAS VEGAS	66
CHICAGO	0
SEATTLE	46

We will save all the really cool Dynamo deep dive stuff and schema design for another dedicated course.

1. Steps for creating a simple Dynamo DB table

- Sign in to the AWS Management Console and in the **Find Services** search box type dynamo and choose **DynamoDB**.
- Make sure you are in the **N. Virginia** region at the top right.
- Click **Create table**.
- For **Table name** type `weather`.
- For **Primary key** type in `sc` (for searchable city) and leave **String** selected.

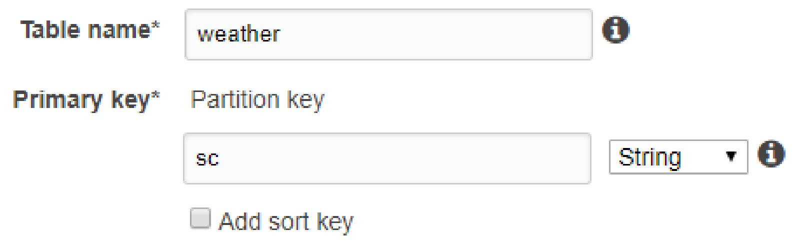
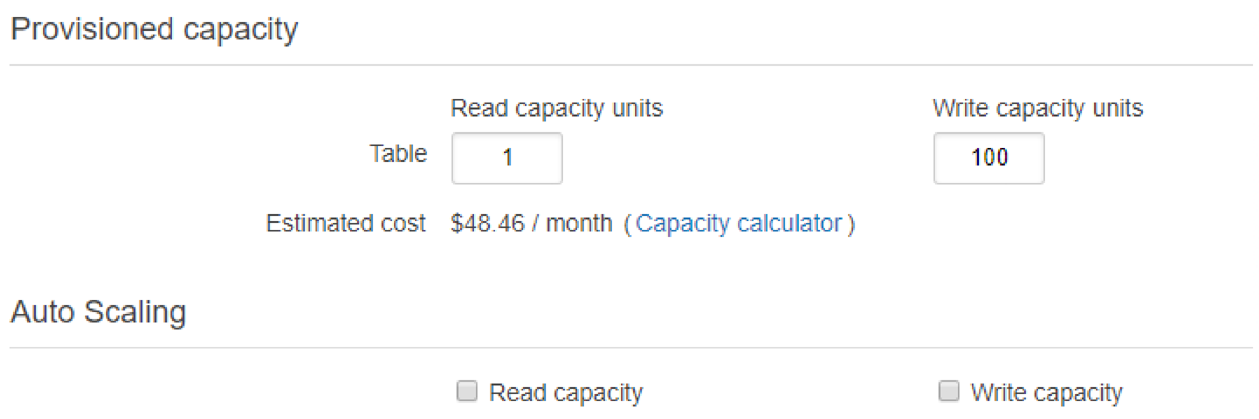


Table name*	weather
Primary key*	Partition key
	sc — String
	☐ Add sort key

- Remove the check for **Use default settings**.
- Under **Auto Scaling** remove the check for **Read capacity** and **Write capacity**.
- Above that under **Provisioned capacity** change the **Read capacity units** to `1` and change the **Write capacity units** to `100`.

We will dial this back down shortly, we only need to have it set to 100 while we seed the table.



Provisioned capacity

	Read capacity units	Write capacity units
Table	1	100
Estimated cost	$48.46 / month (Capacity calculator)	

Auto Scaling

☐ Read capacity ☐ Write capacity

- Leave **DEFAULT** selected under **Encryption At Rest**.
- Click **Create**. Wait ⏰ and verify the **Table status** becomes **Active**.

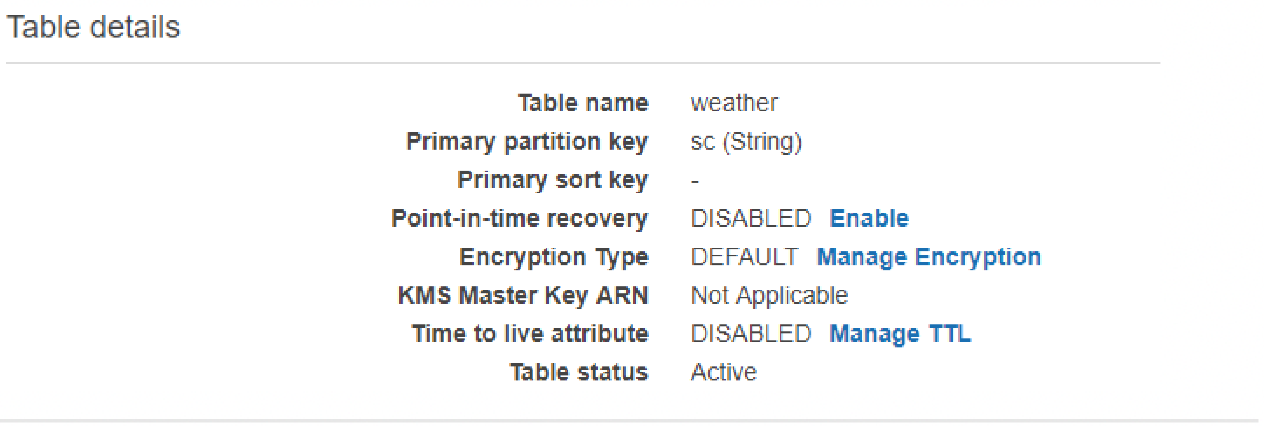


Table details

Table name	weather
Primary partition key	sc (String)
Primary sort key	-
Point-in-time recovery	DISABLED **Enable**
Encryption Type	DEFAULT **Manage Encryption**
KMS Master Key ARN	Not Applicable
Time to live attribute	DISABLED **Manage TTL**
Table status	Active

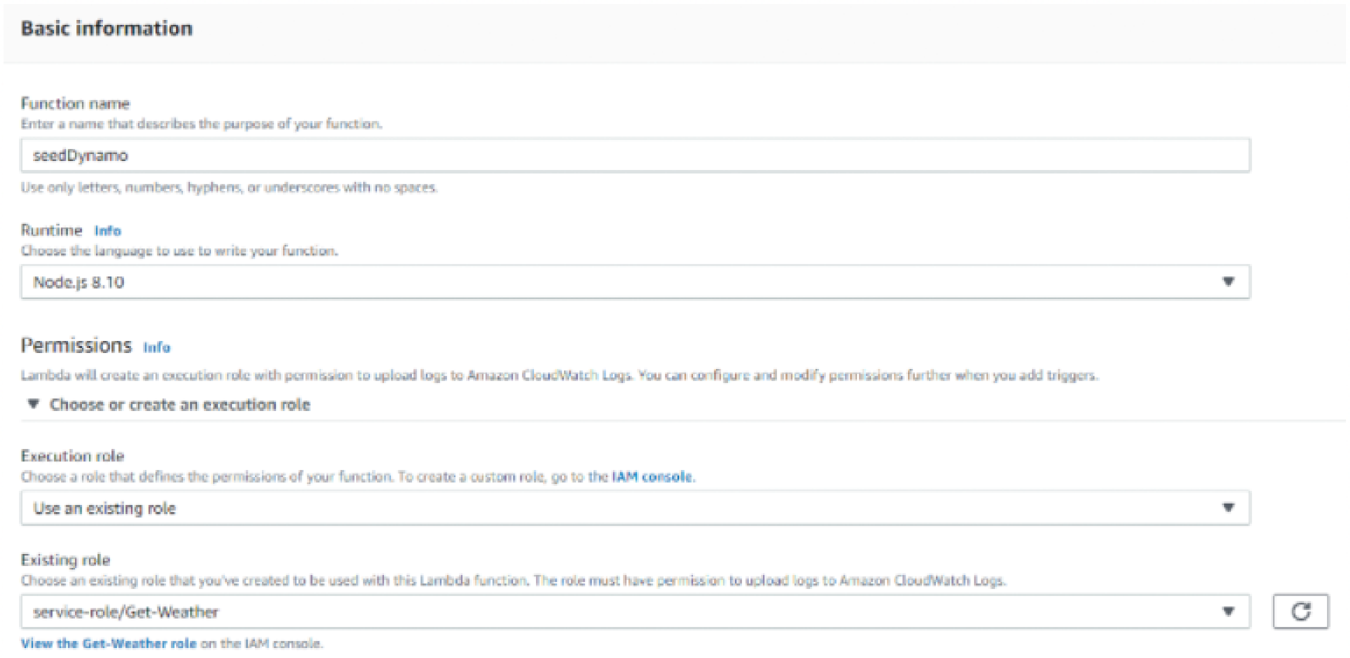
Sometimes it takes a few minutes before the table is active. We must wait for it to say active (keep refreshing) before we try and add data to it.

Once you have it **Active** we can create a Lambda function that will take a provided CSV file and parse it and throw it into Dynamo. That way we have weather data in our database and should be able to do basic queries on it, getting the temperature for that city.

This CSV file is not live data, so in a way it feels like we are cheating. However in the real world you would likely update Dynamo in real time based upon hitting a third party API. This CSV data is fine for our purposes though. At least each city has a different temperatures now, so this will help our website "fake it" pretty well.

2. Steps for seeding the `weather` table from a CSV

- Click **Services** and type lambda in the **Find Services** search box and choose **Lambda**.
- Click **Create function**.
- For **Function name** type in `seedDynamo`.
- Leave **Node.js 8.10** for **Runtime**.
- For **Execution role** select **Use an existing role**.
- For **Existing role** select our **service-role/Get-Weather**.



Basic information

Function name
Enter a name that describes the purpose of your function.
seedDynamo
Use only letters, numbers, hyphens, or underscores with no spaces.

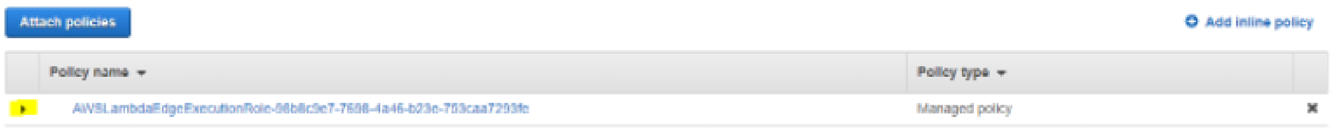
Runtime Info
Choose the language to use to write your function.
Node.js 8.10

Permissions Info
Lambda will create an execution role with permission to upload logs to Amazon CloudWatch Logs. You can configure and modify permissions further when you add triggers.
▼ Choose or create an execution role

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.
Use an existing role

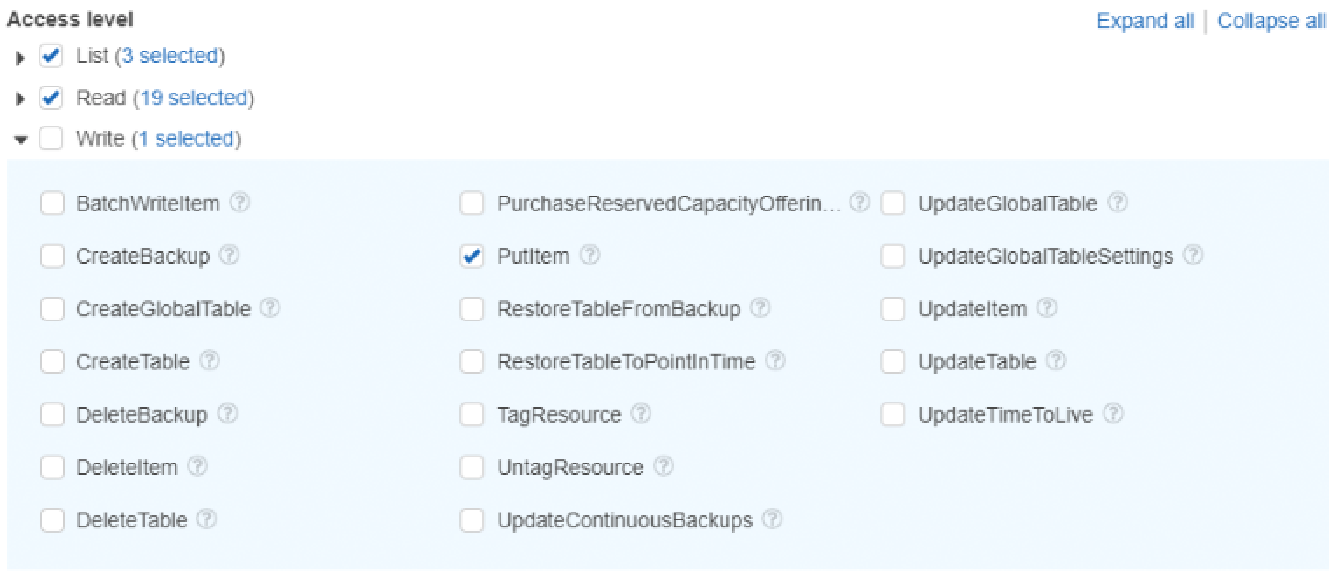
Existing role
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.
service-role/Get-Weather
View the Get-Weather role on the IAM console.

- Click **View the Get-Weather role** on the IAM console. This will pivot to the IAM console.
- Expand the policy by clicking on the drop down arrow.



	Policy name	Policy type	
▸	AWSLambdaEdgeExecutionRole-98b8c9e7-7698-4a46-b23e-703caa7293fe	Managed policy	✖

- Click **Edit policy**.
- Click **Add additional permissions**. As Lambda will need to write to DynamoDB.
- Click **Choose a service**.
- Type dynamo in the **Find a service** search box and choose **DynamoDB**.
- Under **Access level** select **List** and **Read**.
- Expand **Write** and choose **PutItem**.

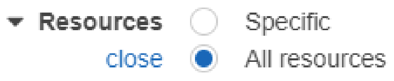


Access level

- [✓] List (3 selected)
- [✓] Read (19 selected)
- [ ] Write (1 selected)

[ ] BatchWriteItem	[ ] PurchaseReservedCapacityOfferin…	[ ] UpdateGlobalTable
[ ] CreateBackup	[✓] PutItem	[ ] UpdateGlobalTableSettings
[ ] CreateGlobalTable	[ ] RestoreTableFromBackup	[ ] UpdateItem
[ ] CreateTable	[ ] RestoreTableToPointInTime	[ ] UpdateTable
[ ] DeleteBackup	[ ] TagResource	[ ] UpdateTimeToLive
[ ] DeleteItem	[ ] UntagResource	
[ ] DeleteTable	[ ] UpdateContinuousBackups	

- Under **Resources** select **All resources**.



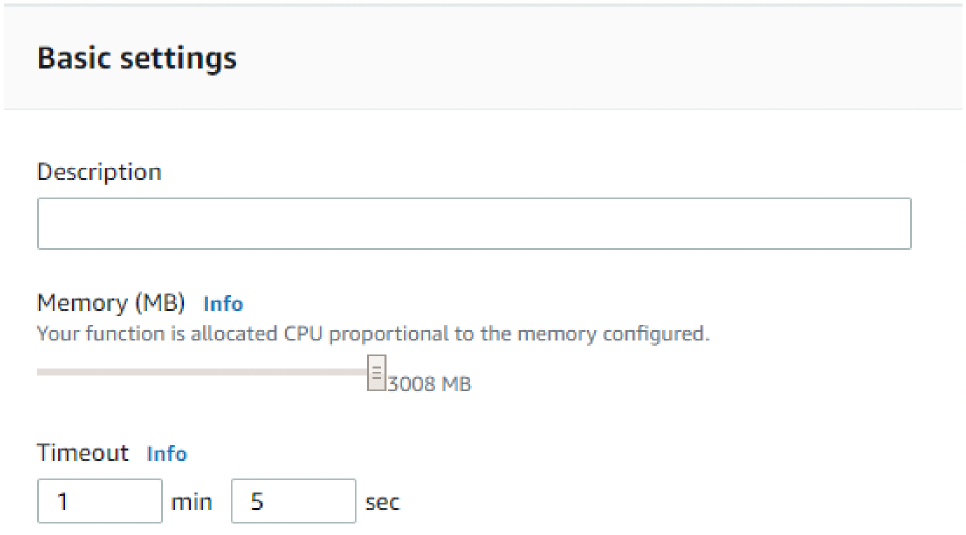
Resources: Specific [✓ All resources]

- Click **Review policy**.
- Click **Save changes**.
- Go back to your **Lambda** tab and click **Create function**.
- Paste the following in the **index.js** tab:

```
exports.handler = function(event, context, callback) {
    var
      AWS = require("aws-sdk"),
      fs = require("fs"),
      item = {},
      some_temp_int = 0,
      params = {},
      DDB = new AWS.DynamoDB;

    AWS.config.update({
        region: "us-east-1"
    });
    fs.readFileSync("cities.csv", "utf8").split('\n').map(function(item_str){
        params.ReturnConsumedCapacity = "TOTAL";
        params.TableName = "weather";
        params.Item = {
            "sc": {
                "S": item_str.split(",")[0]
            },
            "t": {
                "N": String(item_str.split(",")[1])
            }
        };
        DDB.putItem(params, function(err, data){
            if(err){
                console.error(err);
            }else{
                //ignore output
            }
        });
    });
 setTimeout(function(){
     callback(null, "ok");
 }, 1000 * 10);
}
```

This code simply reads a CSV file parses it, and inserts the data into the table. When it is done, it exits.

- Scroll down to **Basic settings** and change **Timeout** to `1` min and `5` sec and set the **Memory** to `3008` MB.



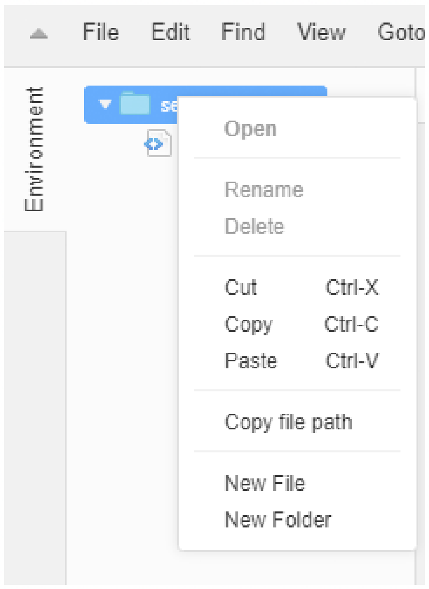
Basic settings

Description

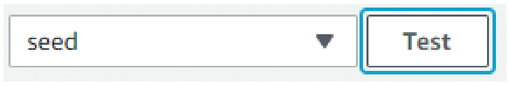
Memory (MB) Info
Your function is allocated CPU proportional to the memory configured.
3008 MB

Timeout Info
1 min 5 sec

- Click **Save** at the top right.
- At the left of the inline code editor where it says **Environment**. Right click the **seedDynamo** folder and click **New File**.

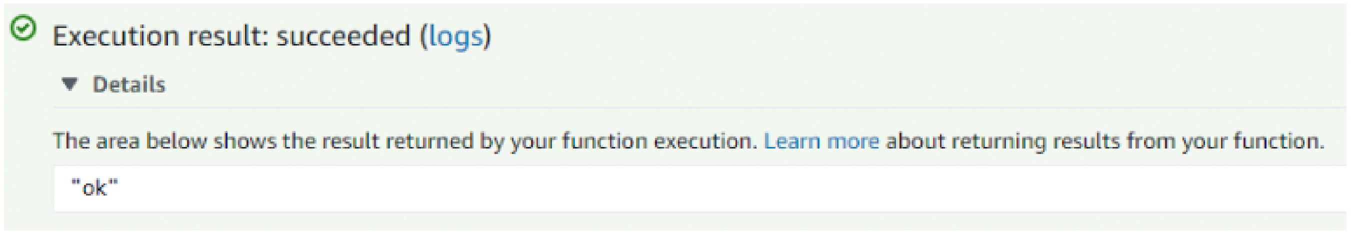


- Name the file `cities.csv`.

- Double click the **cities.csv** file and it will open in a new tab.
- Copy and paste the contents from the **cities_template.md** file which is located in the website zip file you downloaded in week 1.
- Click **Save**.
- Click **Test**. Any test payload we provide will trigger the function, which is all we need to do. The payload you provide is irrelevant.
- For **Event template** leave the **Hello World** setting.
- Under **Event name** type in `seed`.
- Click **Create**.
- Once its populated: 

- Click **Test**. This will populate our **weather** table with the list of cities from our csv file.

This can take up to a full minute, before you see this in the Lambda console.



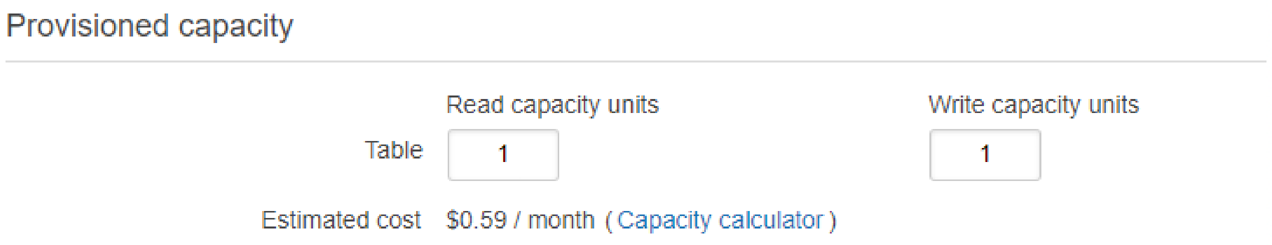
Execution result: succeeded (logs)

Details

The area below shows the result returned by your function execution. Learn more about returning results from your function.

```
"ok"
```

- We want to verify that our DynamoDB table has been populated properly. We also want to change the capacity back to `1` . **As you don't want to be charged for 100WCUs on an ongoing basis**
- Click on **Services** type in dynamo in the search box or choose **DynamoDB** from the **History**.
- Click **Tables** and click our **weather** table.
- Select the **Capacity** tab and under **Provisioned capacity** set the **Write capacity units** to `1`.



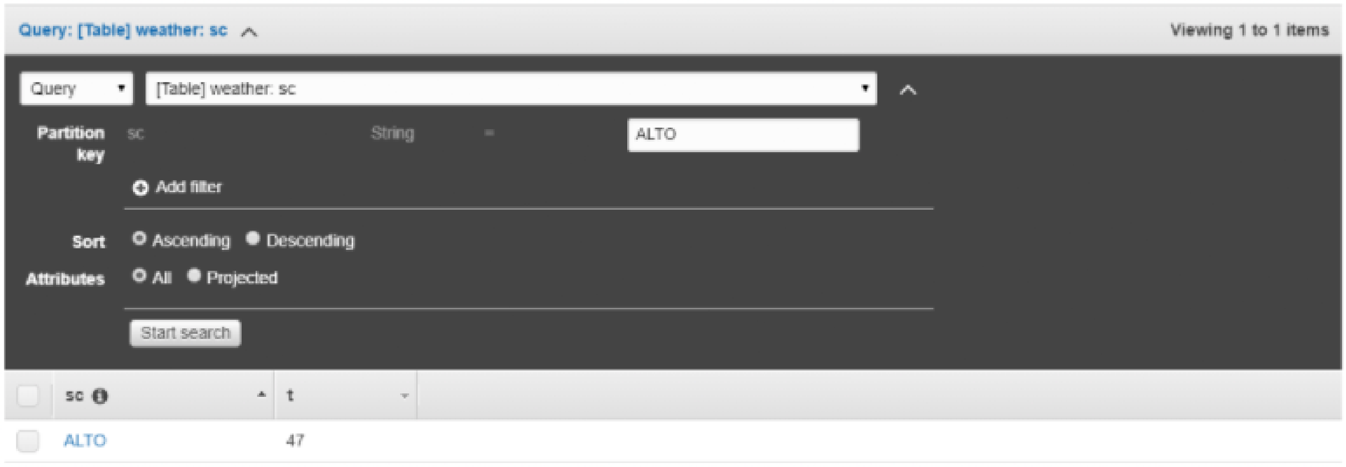
Provisioned capacity

	Read capacity units	Write capacity units
Table	1	1
Estimated cost	$0.59 / month (Capacity calculator)	

- Click **Save**.

Now let's check our table items.

- Click the **Items** tab.
- Change **Scan** to **Query** and enter **ALTO** as the **value**. Then click **Start search**.



sc	t
ALTO	47

- We can see the query returned the correct value **47** from the table.
- We can now remove our **seedDynamo** function.
- Click **Services** and choose **Lambda** from the **History** list.
- Select `seedDynamo` under **Functions**.
- Click **Actions** and **Delete**. On the pop-up window once again click **Delete**.

📔 You notice we cleaned up after ourselves by deleting that seed function. We recommend at the end of the course that you remove all your created assets otherwise costs (albeit small) will occur if you go outside of the free tier.

Ok we have a database, and we have tested that we can query it. Now lets change the Lambda function mock code to something a little more interesting. We will make our Lambda function collect the city that is passed (like before), and instead of just returning it with a random temperature, it is going to issue a request to DynamoDB. Passing the city and hopefully getting a temperature back which it can use to send all the way though API Gateway and back to the users browser.

3. Steps to adjust the Lambda function code so it will query DynamoDB

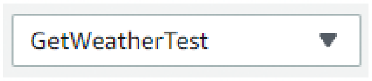
- We should be in our **Lambda** console already.
    - But if you somehow browsed away from it. Click **Services** and search for lambda or choose **Lambda** from the **History**,
- Click our **get_weather** function.
- In the inline code editor replace the contents of **index.js** with:

```
function handler(event, context, callback){
    var
        AWS = require("aws-sdk"),
        DDB = new AWS.DynamoDB({
            apiVersion: "2012-08-10",
            region: "us-east-1"
        }),

        city_str = event.city_str.toUpperCase(),
        data = {
            city_str: city_str,
            temp_int_str: 72
        },
        response = {},
        params = {
            TableName: "weather",
            KeyConditionExpression: "sc = :v1",
            ExpressionAttributeValues: {
                ":v1":{
                    S: city_str
                }
            }
        };

    DDB.query(params, function(err, data){
       var
          item = {},
            response = {
              statusCode: 200,
              headers: {},
              body: null
          };
        if(err){
            response.statusCode = 500;
            console.log(err);
            response.body = err;
        }else{
            // console.log(data.Items[0]);
            var data = data.Items[0];
            if(data && data.t){
                console.log(data.sc.S + " and " + data.t.N);
              item = {
                    temp_int:Number(data.t.N),
                    city_str: data.sc.S
              };
            }else{
                item = {
                  city_str: event.city_str
                  //when we don't return a temp, the client can say city not found
              };
            }
        }
        response = item;
       // console.log(response);
        callback(null, response);
    });
}
exports.handler = handler;
```

This code takes the city and passes it to Dynamo as the partition key. This enables the Dynamo SDK to find the right temperature and return it along with the city.

- Change the **Timeout** to **15** seconds
- Click **Save** at the top right.

Let's create a test case for it, and ensure we can get a different temp for each city.

- We can use our **GetWeatherTest** case. Click the drop down arrow 

- Choose **Configure test events** and change to a desired city. For example:

```json
{
  "city_str": "DENVER"
}
```

- Click **Save**.
- Click **Test**.
- We should now see the following data returned, and temp is coming from our **weather** table:

```json
{
  "temp_int": 38,
  "city_str": "DENVER"
}
```

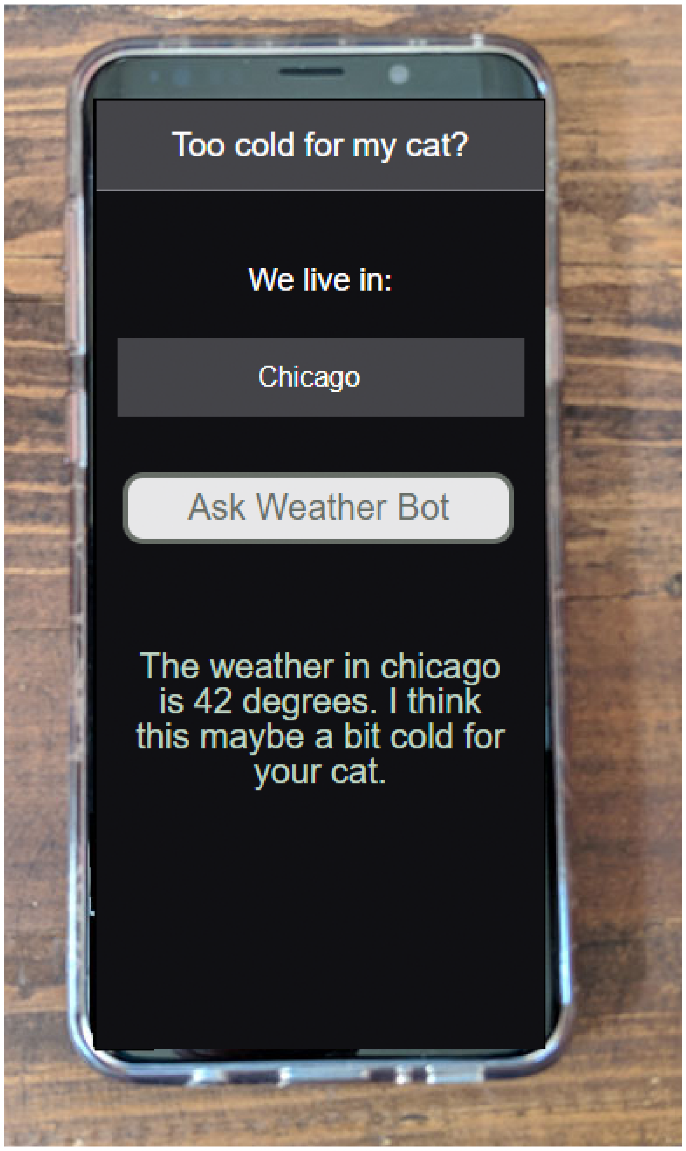
- Feel free to play with the **GetWeatherTest** event and type in different cities. *Use Capitals*

```json
{
  "city_str": "PITTSBURGH"
}
```

Should return:

```json
{
  "temp_int": 78,
  "city_str": "PITTSBURGH"
}
```

- Keep in mind the city must be in our **cities.csv** file or it will simply return **city_str**. Which is a cue to the front end website to say something like: "No city found, please try again".
- We can now visit our website (i.e your CloudFront URL)
- Now when we choose a city and click **Ask Weather Bot** we should get data returned to us from our **weather** DynamoDB table:
- Depending upon the weather temp, the JavaScript in the front end website will add a 2 cent comment, suggesting that it is either too hot, too cold, or just right for your cat.



The weather in chicago is 42 degrees. I think this maybe a bit cold for your cat.

We are faking it pretty well right now, we have a server-less data driven text weather app running in a global content delivery network for low latency.

The question is. Can we take this to the next level?

Can we integrate the cool functionality of LEX into this, and get this website chatting to us with voice in stead of text. Can it start to understand phrases that we didn't teach it, and still hold down a conversation (I am using the term conversation loosely here!).

Ok see you in week 4, where we will stand on the shoulders of our LEX bot we created in week 1 and tie it into all these other services that you have spent the last three weeks learning.

Exercise goal checklist

1. ~~Create a simple chatbot using the lex console.~~
2. ~~Upload our website to S3.~~
3. ~~Create a content delivery network and lock down S3.~~
4. ~~Build an API gateway mock with CORS.~~
5. ~~Build a Lambda mock, use IAM, push logs to CloudWatch.~~
6. ~~Create and seed a database with weather data.~~
7. ~~Enhance the lambda, so it can query the database.~~
8. ~~Play with your new text based data driven application.~~
9. Create a LEX proxy using Lamba.
10. Enhance API gateway to use the LEX proxy.
11. Play with your new voice web application.