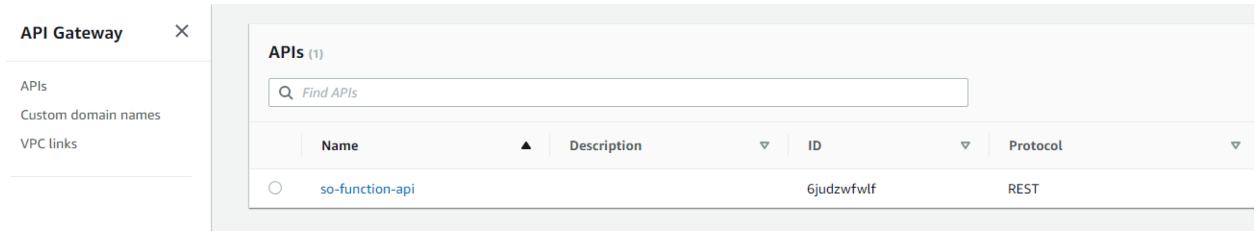
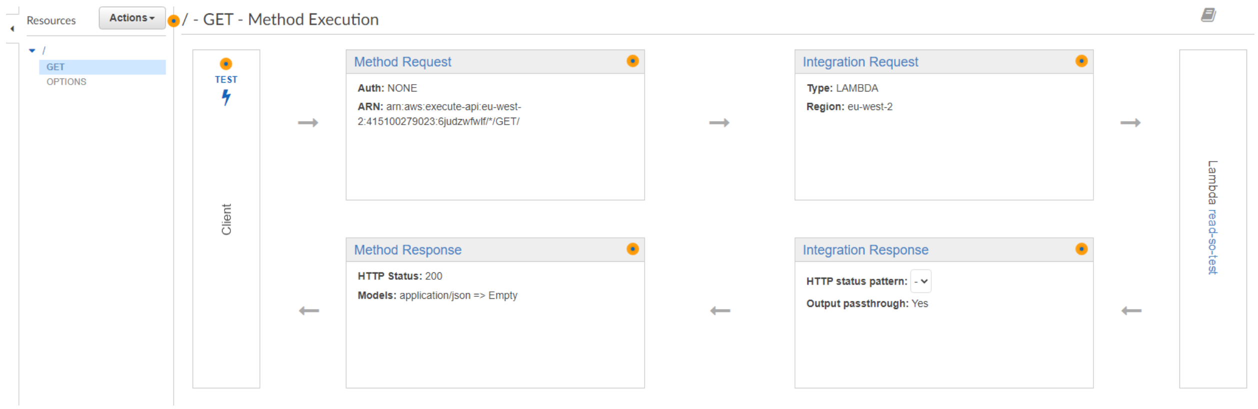
AWS content documentation

The back end of this project is hosted on AWS, and as such cannot be zipped and installed locally, it is permanently cloud hosted. In order to access this service you will require an internet connection, or else the database cannot be accessed.
Because it cannot be installed locally and part of a personal AWS account, the code and configurations of the AWS architecture I used is documented here instead of in the repo. The node.JS code from the Lambda functions is documented, but also out of context - it is included here with the context of the AWS architecture that was used to deliver the back end functionality as part of the development process.

The basic flow of data is as follows;
The front end sends a basic http request via a RESTFUL API created in API Gateway. API Gateway then routes this based on the request - a GET request will be routed to fire up a Lambda following its configuration, which will take parameters passed into the API and refactor them to a request expected by DyanmoDB, and then pass them to the database connected to the Lambda function. The database sends its return to the Lambda, which then hands it forwards into the API Gateway and spins down - the API Gateway hands the return back as a JSON object, which is then handled by the front end.



Name	Description	ID	Protocol
so-function-api		6judzwfwlf	REST



Method Request
Auth: NONE
ARN: arn:aws:execute-api:eu-west-2:415100279023:6judzwfwlf/*/GET/

Integration Request
Type: LAMBDA
Region: eu-west-2

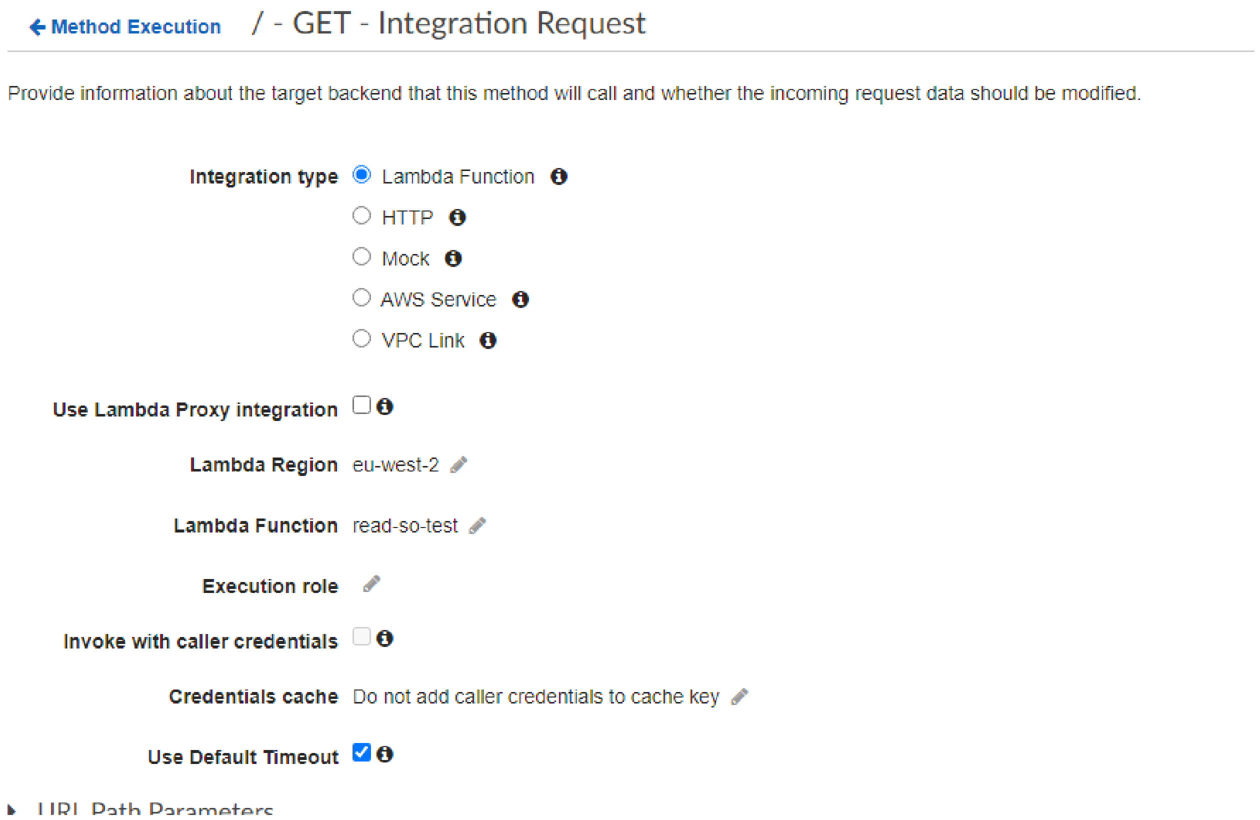
Method Response
HTTP Status: 200
Models: application/json => Empty

Integration Response
HTTP status pattern: -
Output passthrough: Yes

Above we can see the API Gateway created to service the “read” requirement of the CRUD database requirements. It is configured to function as a REST API with a GET request parameter to handle incoming requests. Further development to satisfy all CRUD requirements would have used other request types, PUT (to create and update) and DELETE (to delete), but development of this was not completed and thus not implemented.
The developed GET request in the second image shows the cycle it takes from the API perspective, invoking the lambda read-so-test after the integration request and before the

integration response. This is an internal tool, handling the power of the API to invoke lambdas through another AWS functionality - IAM roles. This layer of security allows us to assign the minimum powers to invocations, functions, services and users to complete their tasks. For this microservice, we only needed to grant the API Gateway permission to invoke a Lambda.

The security of this particular API has been left vulnerable - CORS permissions allows us to reject any call to the API that is not sent from a specific URL, but as this submission is installed and hosted locally on other machines, and because the API URL has not been directly exposed to any third parties, there is a requirement that CORS allowance be set to a “wildcard (*)” state, allowing all requests through. Because the URL has not been exposed, this danger is directly mitigated by the remote chances of it being invoked at random. It is worth noting that had the work item created to host the webapp been completed, then this permission level could be altered to only allow that particular URL to invoke the API - in a production environment this would be the first basic layer of security on the API, but here it has been intentionally left out.

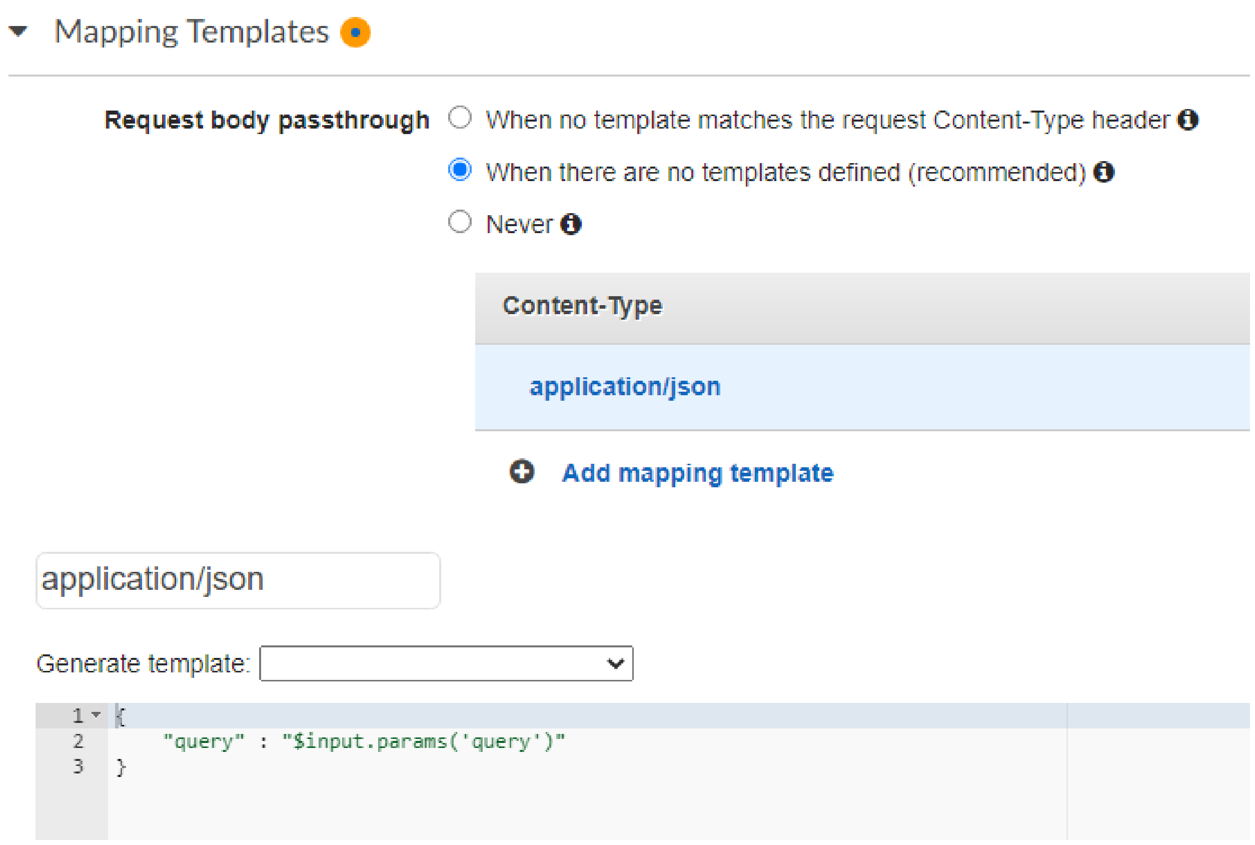


/ - GET - Integration Request

Provide information about the target backend that this method will call and whether the incoming request data should be modified.

Integration type	[✓ Lambda Function] HTTP Mock AWS Service VPC Link
Use Lambda Proxy integration	☐
Lambda Region	eu-west-2
Lambda Function	read-so-test
Execution role	
Invoke with caller credentials	☐
Credentials cache	Do not add caller credentials to cache key
Use Default Timeout	✓

Here we are looking at the integration request details, where we configure which Lambda function we invoke to service the API. Selecting Lambda Function allows us to autofill the rest of the requirements, able to select read-so-test from a drop down menu because we are working in the same eu-west-2 (London) region of AWS. Below, we also create a map template to handle the data that is passed forwards to the Lambda as it is invoked following a basic key value json object template, which we will handle within the Lambda itself.



Request body passthrough
( ) When no template matches the request Content-Type header
[✓ When there are no templates defined (recommended)]
( ) Never

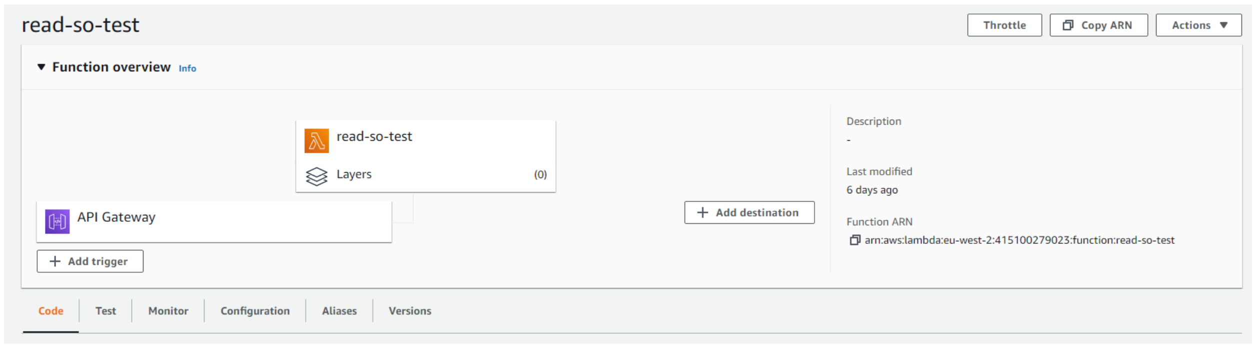
Content-Type
application/json

application/json

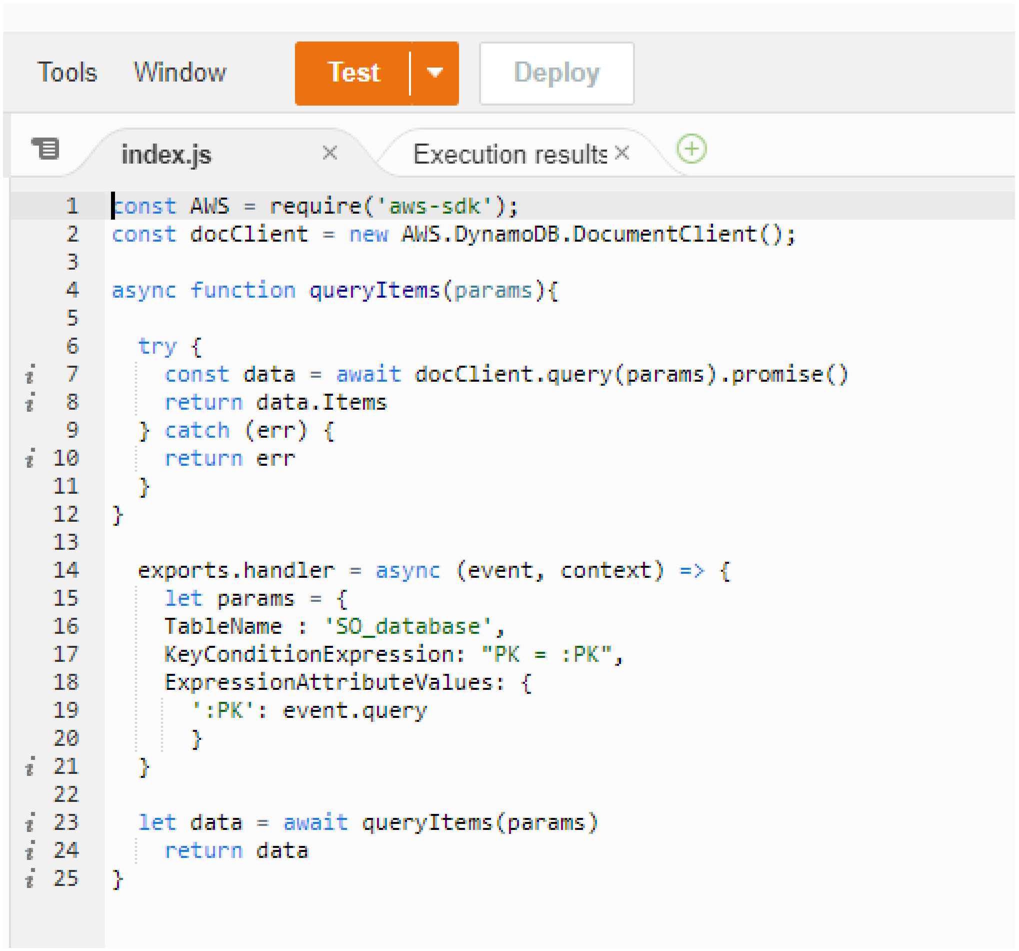
Generate template:

```
{
    "query" : "$input.params('query')"
}
```

Below we can see the API read-so-test that is connected to our API Gateway, and the list of trigger events that cause the API to be spun up. Because this is cloud hosted, this microservice code does not exist outside of it being used by the API Gateway - each invocation of the API Gateway creates this Lambda, runs the code, then deletes the instance as required. Because of this there is a requirement to keep Lambda code as short and as functional as possible.

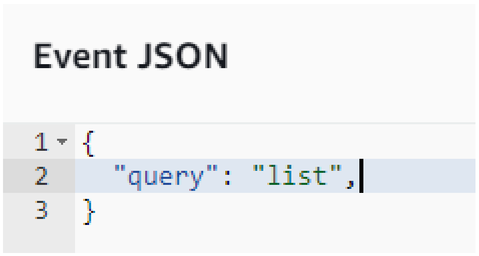


Description
-
Last modified
6 days ago
Function ARN
arn:aws:lambda:eu-west-2:415100279023:function:read-so-test

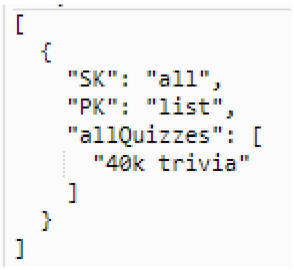


```
const AWS = require('aws-sdk');
const docClient = new AWS.DynamoDB.DocumentClient();

async function queryItems(params){

  try {
    const data = await docClient.query(params).promise()
    return data.Items
  } catch (err) {
    return err
  }
}

  exports.handler = async (event, context) => {
    let params = {
    TableName : 'SO_database',
    KeyConditionExpression: "PK = :PK",
    ExpressionAttributeValues: {
      ':PK': event.query
      }
  }

  let data = await queryItems(params)
    return data
}
```

Here we can see the node.JS code written within the Lambda, using AWS-sdk and DynamoDB libraries and syntax to call our Dynamo table and handle the API call data to create a search query to service the Read requirement of CRUD. “event.query” is the data passed through as “query” from the API event handler above, returned as “data.Items” to the front end where this is processed.
The dev testing of this API is a very simple test to see if inputting specific parameters returns the expected results from the database. The data is input using the same formatting as the above event handler map template;



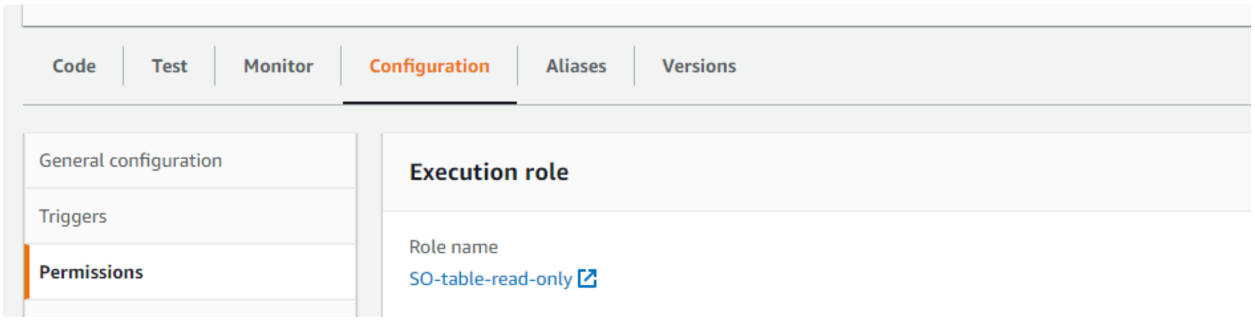
```
{
  "query": "list",
}
```



```
[
  {
    "SK": "all",
    "PK": "list",
    "allQuizzes": [
      "40k trivia"
    ]
  }
]
```

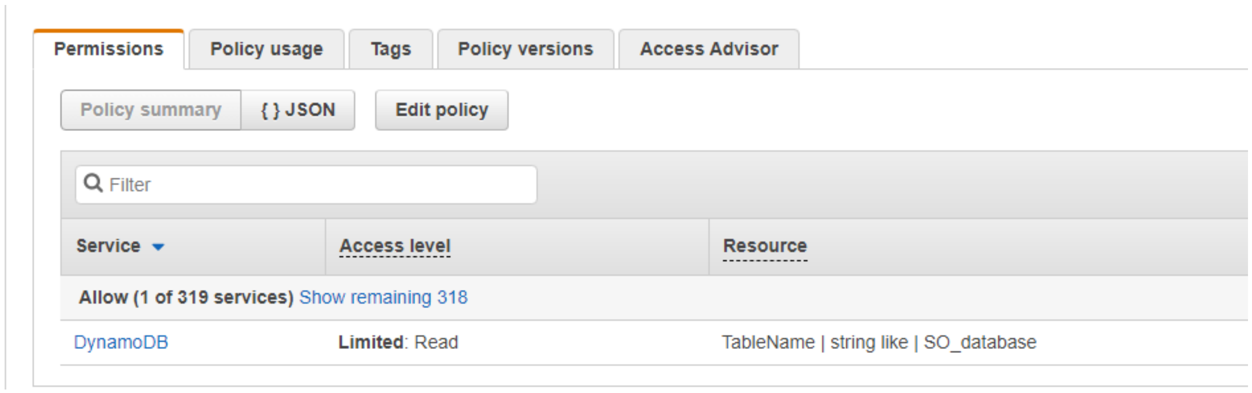
This also shows us the parameters we need to pass forward from the front end through the API to get the results, and the exact result returned, which gives us the information we need to sort through the return and populate our front end to make subsequent calls for further data, and what we expect in an error-free API call to return the front end. It also shows us that from the

Lambda the code is working as expected, allowing us to rule out the Lambda and the Dynamo table as causes of errors. This was very useful during development.



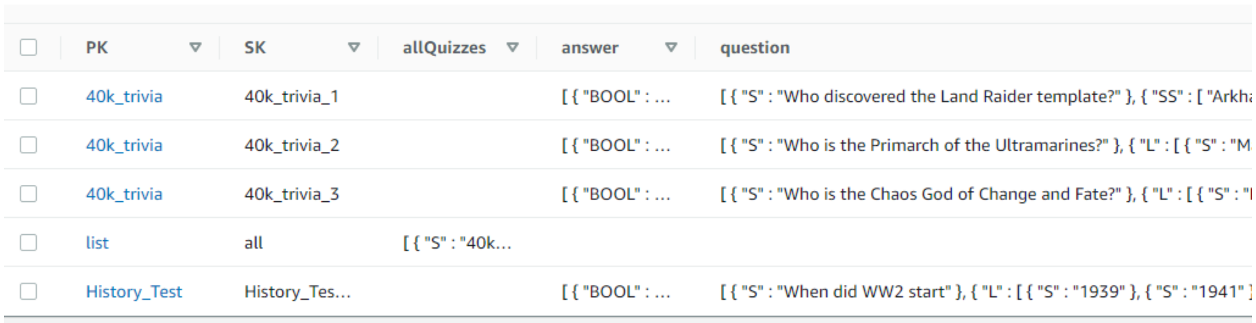
Execution role

Role name
SO-table-read-only



Service	Access level	Resource
Allow (1 of 319 services) Show remaining 318		
DynamoDB	**Limited**: Read	TableName | string like | SO_database

In order to limit the power of a Lambda in case of malicious access to the API, we can further restrict what is possible for the code to perform by applying limits to the IAM access permissions of the Lambda - moreso than with the API Gateway this is important as being node.JS the Lambda is potentially more powerful than the API Gateway. This Lambda's only function is to read the database and return the data it finds, so the permission levels for it are capped at that. To do this a specific IAM role was created for the Lambda. This can be limited not only to just Dynamo, but to a specific table. It can be written using JSON format, but AWS has tools to create them using the GUI while remaining within the AWS architecture, as I did for this project.



PK	SK	allQuizzes	answer	question
40k_trivia	40k_trivia_1		[{ "BOOL" : …	[{ "S" : "Who discovered the Land Raider template?" }, { "SS" : ["Arkh
40k_trivia	40k_trivia_2		[{ "BOOL" : …	[{ "S" : "Who is the Primarch of the Ultramarines?" }, { "L" : [{ "S" : "M
40k_trivia	40k_trivia_3		[{ "BOOL" : …	[{ "S" : "Who is the Chaos God of Change and Fate?" }, { "L" : [{ "S" : "I
list	all	[{ "S" : "40k…		
History_Test	History_Tes…		[{ "BOOL" : …	[{ "S" : "When did WW2 start" }, { "L" : [{ "S" : "1939" }, { "S" : "1941" }

Finally we have the database built using the datastructures outlined previously. With DynamoDB each Item is created to its corresponding data entity wether it exists in the table or not - the table is expanded to accommodate new fields in the item range. The partition and sort keys require data (here they are simply PK and SK using Dynamo best practice). Our data items use Lists to place information within arrays so we can map through them and populate tables to create our quiz on the front end, as per requirements. Here you can also see the list of boolean values used to store the correct answers - each answer had a corresponding boolean of true for correct and false for wrong stored in a separate array, connecting them would be done in the front end computationally.