First ClassJS
Modern Javascript from A to Z

May 30, 2019 • AWS / Node.JS

Serverless CRUD API using AWS Lambda, DynamoDB, API Gateway and Node.JS

Posted by Viktor Borisov

The purpose of this tutorial is to show you how to create your first serverless API using Amazon Web Services(AWS) Lambda, DynamoDB, API Gateway for API exposure and of course Node.JS. My main goal is to introduce you to the basics of using AWS, not the best practices to write Node.JS code.

You can see the complete code in First Class JS – GitHub(link).

In order to understand the code, we will explore it file by file, step by step.

Here is my implementation plan:

- Configure **AWS** – Create **Lambda** function with **API Gateway** and **DynamoDB** database table creation
- Setup new **Node.JS** project using **Serverless Express** and implement basic routes
- Automate the deploy process using **AWS CLI**
- Implement local development capabilities using **Docker Compose** (for easier development and testing)

Configure AWS – Create Lambda function with API Gateway and DynamoDB database table creation

Before continue, you will have to register to AWS(if you haven't already) – https://aws.amazon.com/free/

We will use the popular Lambda service as a back-end. The cool thing about it is that we don't have to care about scaling and other server administration/management things. Just "deploy" your code and you are ready to go. Ah .. and one more thing – you are paying only for the compute time you consume. Don't worry, there is plenty of it for free 🙂

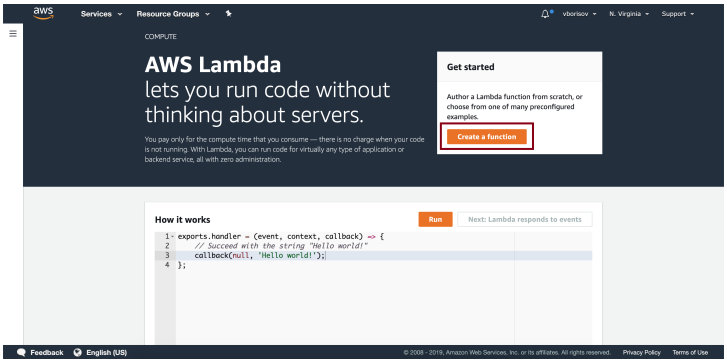
Go to https://console.aws.amazon.com/lambda

- Mock/Fake Backend API in Angular – How To
- How to convert string to number in JavaScript (BE CAREFUL!)
- Serverless CRUD API using AWS Lambda, DynamoDB, API Gateway and Node.JS
- Check if Object is empty in Javascript
- Under The Hood: Arrays in JS

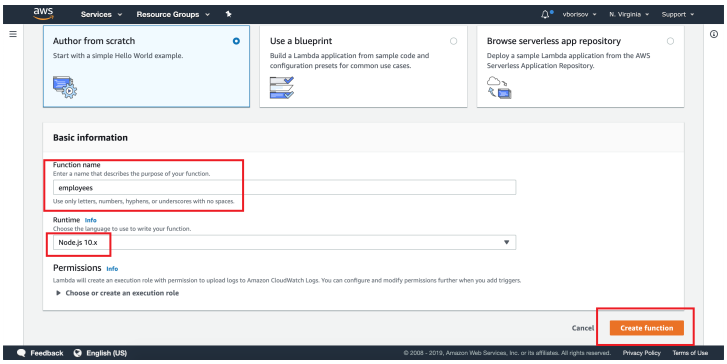
Categories

- Algorithms
- Angular
- AWS
- JavaScript
- Node.JS

Choose the region from the top right and click **Create a function**



Fill the function name, I will use 'employee' for this example, Runtime – Node.js 10.x.



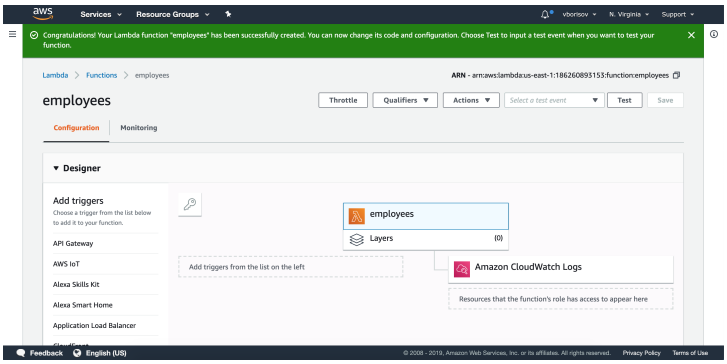
Leave the Permissions field as it is, we will get back later to it to add permission our Lambda to access DynamoDB. Until then, our function will use the basic role which has really limited access and will be able to only upload logs to CloudWatch.

Click **Create Function**. It will take a few seconds before a success message "*Congratulations! Your Lambda function "employees" has been successfully created. You can now change its code and configuration. Choose Test to input a test*

event when you want to test your function."

appear.

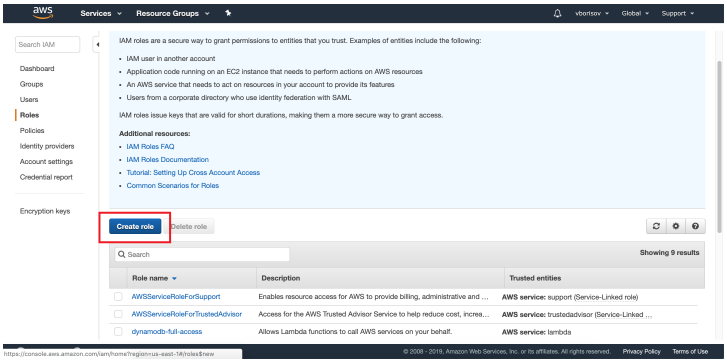
We will also be redirected to the newly created lambda.



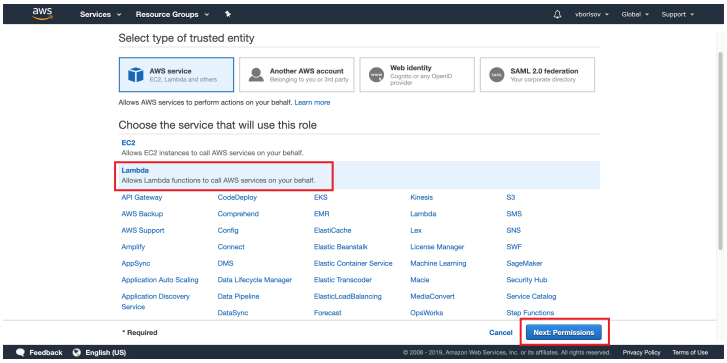
Now, when our function is created, let's switch for a moment to **IAM**(Identity and Access Management) and create a role for it. We will need it, as I already mentioned, to grant access to **DynamoDB** – the database we will use.

Go to https://console.aws.amazon.com/iam

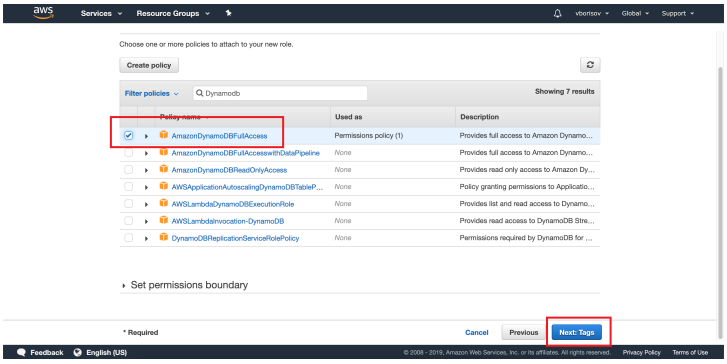
Click **Roles** and then **Create Role**.



Choose the service that will use this role – in our case **Lambda** and click **Next: Permissions**

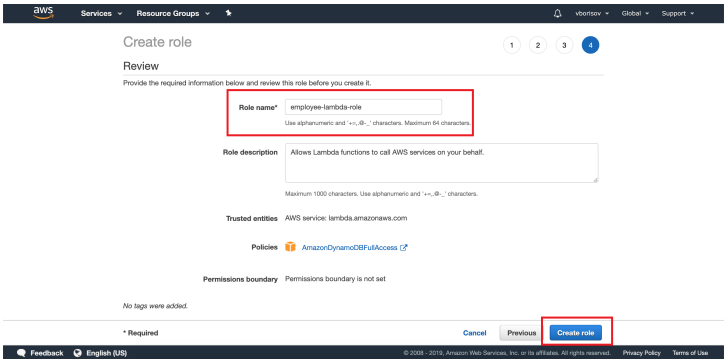


Here, we can create our own custom policy or use the already available ones. The policies are basically rules in JSON format that tells the role what permissions should be given to the service attached to it. For our example here, we will use the already available **AmazonDynamoDBFullAccess** policy.

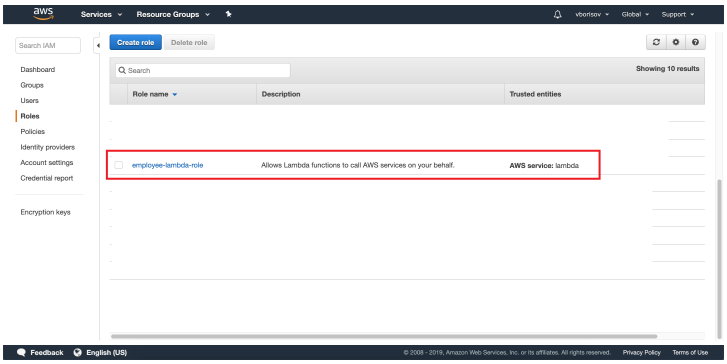


Click **Next** and again **Next** and you should view the **Review** part.

Fill the desired role name, something like ‘employee-lambda-role’ and click **Create Role**

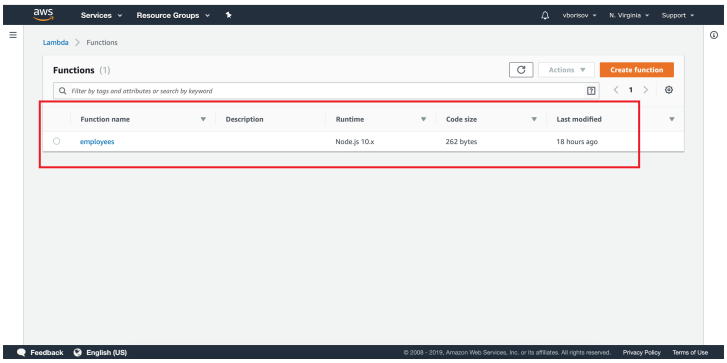


The role should be created and available in the list of roles available in **IAM**.



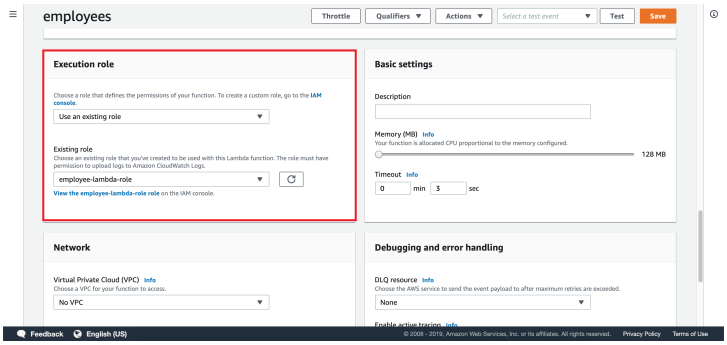
Now, we can get back to our lambda and assign this role to it.

Go to https://console.aws.amazon.com/lambda and choose your function.



You will be redirected to the specific Lambda page. Scroll down to **Execution role** and choose

the role you want to use, in my case ‘employee-lambda-role’.

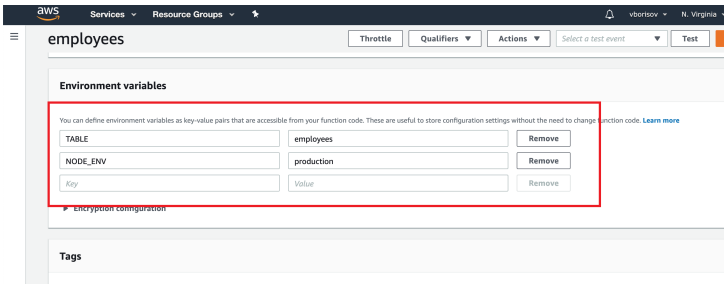


We will do one more thing before saving it.

Go to the “Environment variables” section and include two variables which we will use later when writing our Node.JS logic.

TABLE: employees // the name of our future database table (will create it soon)

NODE_ENV: production // the environment, let’s call it ‘production’. This will help us to identify if it is local or serverless instance of the server

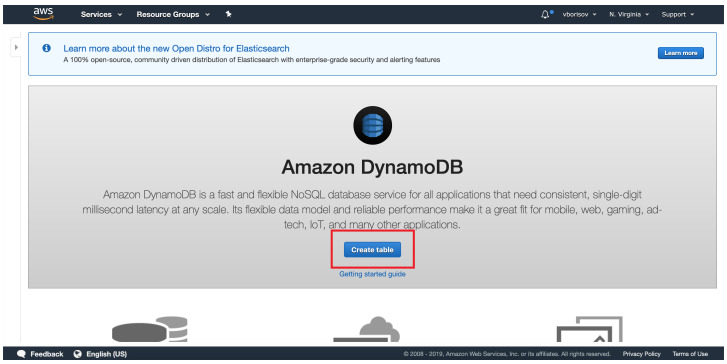


And click **Save**.

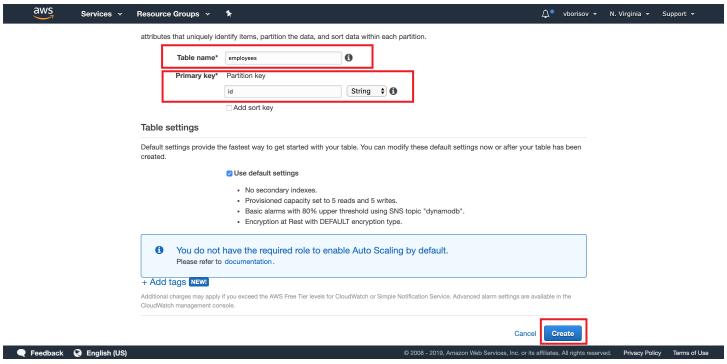
Okay, so far so good. It's time to configure **DynamoDB**.

Go to https://console.aws.amazon.com/dynamodb

Click **Create Table**



Fill the table name to **employees** (if you are following this tutorial) and Primary key **id** with type string.



And click **Create**.

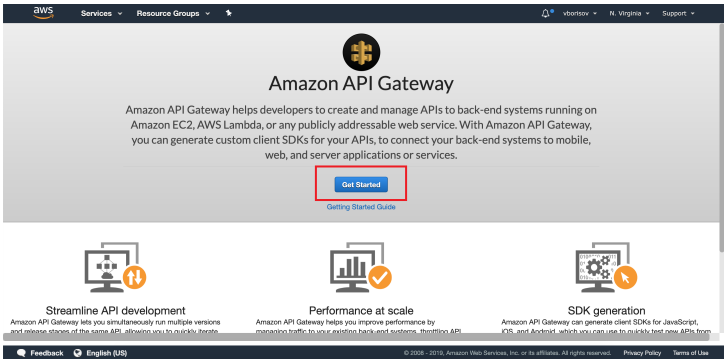
When the table is created you should be redirected to the table management route.

This means that your table creation was successfully. We are done with the **DynamoDB**

configuration.

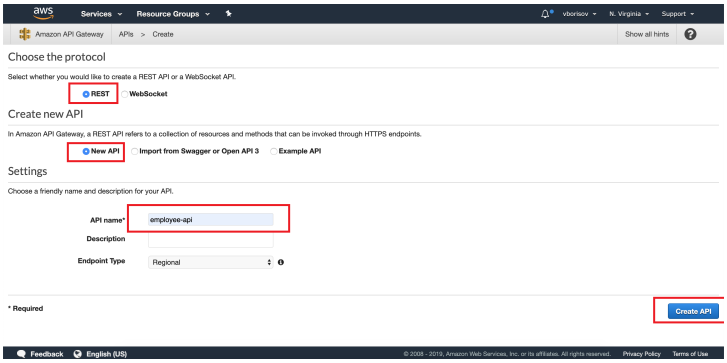
Now it's time to create **API Gateway** and connect it to the **Lambda** we have created earlier.

Go to https://console.aws.amazon.com/apigateway and click **Get Started**.



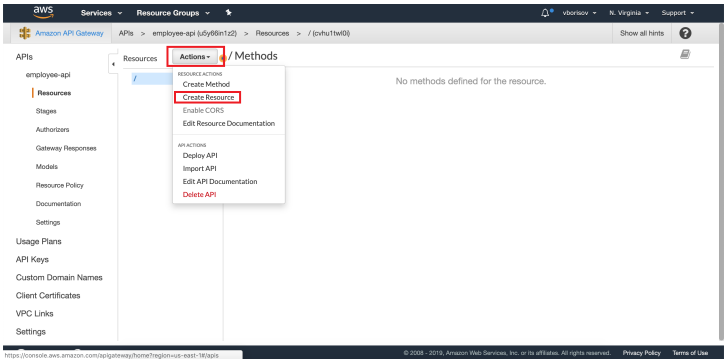
Choose the protocol of you API to be REST.

We will create a brand new API by choosing New API and the API name will be 'employee-api'. See the image below:



Click **Create API** and soon you will be redirected to the newly created API.

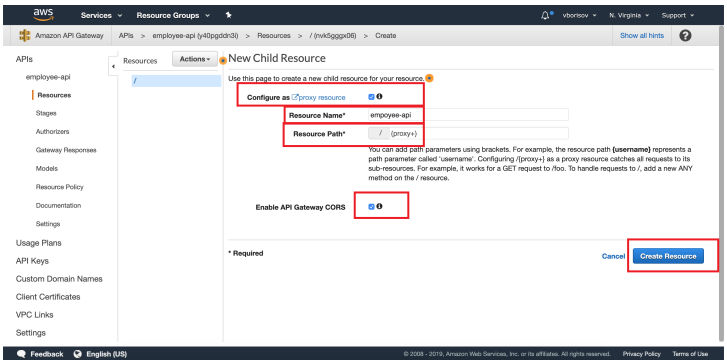
Got to the actions tab and choose **Create Resource**.



Configure as proxy resource have to be checked – this way we will handle the routes in our Lambda function and there will be no need to manually add every endpoint in the gateway every time we create one.

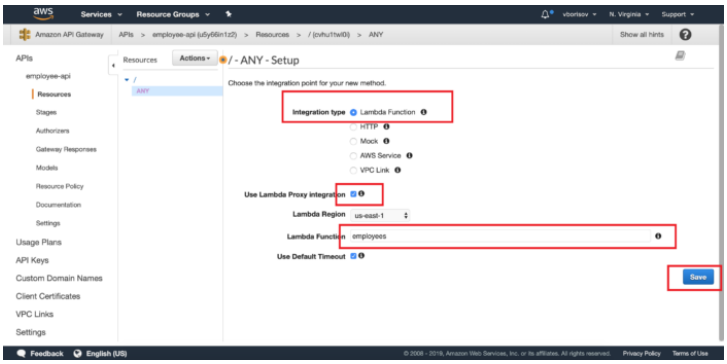
Resource name – 'empoyee-api' and **Resource path** – {proxy+} (you can find more information below the field what {proxy+} means).

Enable API Gateway CORS is not required, but I suggest you to also check it. This way you can configure later the origins you want to have access to your resource, methods and etc.



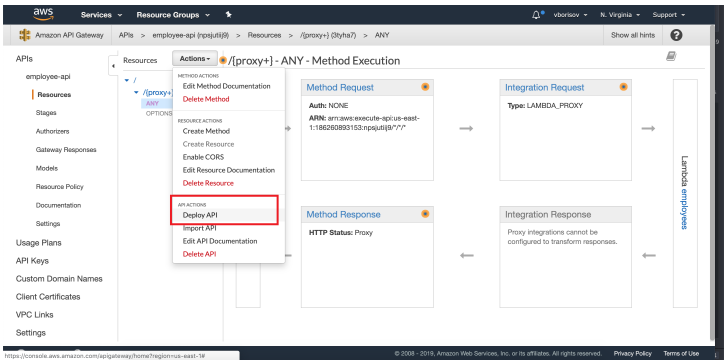
And click **Create Resource**

Then specify the Lambda function you want to connect to your newly created API Gateway resource.



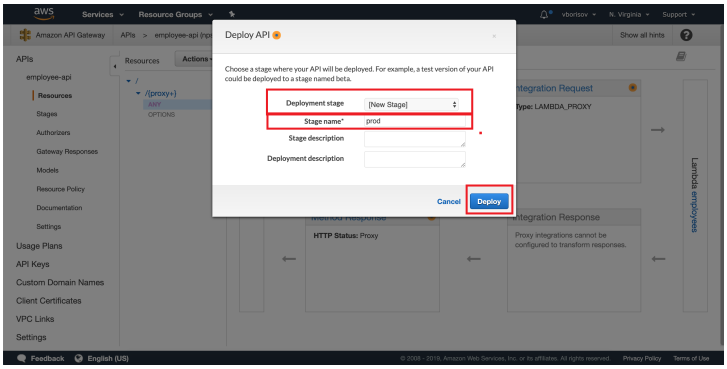
And the last thing is to deploy this resource.

Click on **Actions -> Deploy**

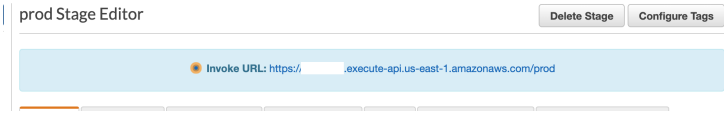


Choose the **Deployment stage -> [New Stage]** and **Stage Name -> 'prod'.**

The others are fields are optional.



Click Deploy and soon you will see your API endpoint, like the one below:



This is your base API url, which we will use from now on to access it.

Well done! This is the initial configuration for our API and very important part of the tutorial. This is the last time we will use the AWS console, from now on the AWS Cli will be our friend for future deploys and configurations. Make sure you configure it before continue.

More information: https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

Now, we can create the application.

Setup new Node.JS project using Serverless Express and implement basic routes

Create a new directory, I will name it **express-serverless-crud.**

Go to that newly created directory and initialize a new Node.JS project.

```
npm init
```

You can leave everything as it is during the creation.

At the end you will be asked to confirm the settings, which will be something like this:

```
{
  "name": "employee-api",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Type **yes** and and press enter. You should see a new **package.json** file available in your directory now.

We will need a few packages. Use the following command to install them:

```
npm i aws-sdk aws-serverless-express cors express uuid --save
```

What we will use each of them for:

aws-sdk – to interact with AWS

express & **aws-serverless-express** – to use the power of express, rather than writing vanilla Node.JS

cors – package to enable cors for as a middleware in express

uuid – to generate a unique id(guid) for the employees

Now, we can create the entry code for our app.

Add a new **app.js** file in the root folder with the following content:

```
const express = require('express')
const app = express();
const bodyParser = require('body-parser');
const cors = require('cors');
const awsServerlessExpressMiddleware = require('aws-serverless-express/middleware')

const routes = require('./routes');

app.use(cors());
app.use(bodyParser.json());
```

```
app.use(bodyParser.urlencoded({
extended: true }));

app.use(awsServerlessExpressMiddlewa
re.eventContext());

app.use('/', routes);

module.exports = app;
```

It's a standard entry file for express applications with and extra middleware

```
awsServerlessExpressMiddleware.event
Context()
```

This middleware is taking care of the **eventContext** object received by the API Gateway and transform the object to something more understandable by express.

You can notice that there is also routes imported in this file(we can not go without routes, right :)). Create a new file called **routes.js** and include the following code there.

```
const AWS = require('aws-sdk');
const express =
require('express');
const uuid = require('uuid');

const IS_OFFLINE =
process.env.NODE_ENV !==
'production';
const EMPLOYEES_TABLE =
process.env.TABLE;

```

```
8. const dynamoDb = IS_OFFLINE ===
true ?
9.     new
AWS.DynamoDB.DocumentClient({
10.         region: 'eu-west-2',
11.         endpoint:
'http://127.0.0.1:8080',
12.     }) :
13.     new
AWS.DynamoDB.DocumentClient();
14.
15. const router = express.Router();
16.
17. router.get('/employees', (req,
res) => {
18.     const params = {
19.         TableName:
EMPLOYEES_TABLE
20.     };
21.     dynamoDb.scan(params, (error,
result) => {
22.         if (error) {
23.
res.status(400).json({ error: 'Error
fetching the employees' });
24.         }
25.         res.json(result.Items);
26.     });
27. });
28.
29. router.get('/employees/:id',
(req, res) => {
30.     const id = req.params.id;
31.
32.     const params = {
33.         TableName:
EMPLOYEES_TABLE,
34.         Key: {
35.             id
36.         }
37.     };
38.
39.     dynamoDb.get(params, (error,
result) => {
40.         if (error) {
```

```
41.
res.status(400).json({ error: 'Error
retrieving Employee' });
42.         }
43.         if (result.Item) {
44.
res.json(result.Item);
45.         } else {
46.
res.status(404).json({ error:
`Employee with id: ${id} not found`
});
47.         }
48.     });
49. });
50.
51. router.post('/employees', (req,
res) => {
52.     const name = req.body.name;
53.     const id = uuid.v4();
54.
55.     const params = {
56.         TableName:
EMPLOYEES_TABLE,
57.         Item: {
58.             id,
59.             name
60.         },
61.     };
62.
63.     dynamoDb.put(params, (error)
=> {
64.         if (error) {
65.
res.status(400).json({ error: 'Could
not create Employee' });
66.         }
67.         res.json({
68.             id,
69.             name
70.         });
71.     });
72. });
73.
74. router.delete('/employees/:id',
(req, res) => {
```

```
    const id = req.params.id;

    const params = {
        TableName:
EMPLOYEES_TABLE,
        Key: {
            id
        }
    };

    dynamoDb.delete(params,
(error) => {
        if (error) {

res.status(400).json({ error: 'Could
not delete Employee' });
        }
        res.json({ success: true
});
    });
});

router.put('/employees', (req,
res) => {
    const id = req.body.id;
    const name = req.body.name;

    const params = {
        TableName:
EMPLOYEES_TABLE,
        Key: {
            id
         },
         UpdateExpression: 'set
#name = :name',

ExpressionAttributeNames: { '#name':
'name' },

ExpressionAttributeValues: {
':name': name },
         ReturnValues: "ALL_NEW"
     }

    dynamoDb.update(params,
(error, result) => {
```

```
108.         if (error) {
109.
res.status(400).json({ error: 'Could
not update Employee' });
110.         }
111.
res.json(result.Attributes);
112.     })
113. });
114.
115. module.exports = router;
```

For now on, skip the IS_OFFLINE variable, we will use it a little bit later when adjusting the project to work with local version of AWS and DynamoDB. This variable will always be false when deployed to AWS as we have included a NODE_ENV to be **production** in the Lambda.

The rest of the code are basic CRUD operations with DynamoDB – Get all employees, Get specific employee, Add employee, Delete employee and Update(Edit) employee. I will not delve in them as I think they are pretty self explanatory.

One more thing needed is the entry file for the **LAMBDA**(yes, it's different than the app.js file we created before). It's a file/code specific for the online version of the app, we will not use it for local development.

Create an **index.js** file with the following content:

```
const awsServerlessExpress =
require('aws-serverless-express')
const app = require('./app')
const server =
awsServerlessExpress.createServer(ap
p)

exports.handler = (event, context)
=> {
awsServerlessExpress.proxy(server,
event, context) }
```

Basically it just proxify the request/response to be compatible with serverless express.

Automate the deploy process using AWS CLI

Okay, now we can take the **node_modules** folder, **index.js**(the entry point of the lamba), **app.js**(the hearth of the application) and **routes.js**(well the routes :)), pack them to zip, go to the lambda page and upload them. Instead of doing this, we will use aws cli to do the job for us.

Go to the **package.json** and include the following three lines in scripts part:

```
  "deploy": "npm run clean && npm
run build && aws lambda update-
function-code --function-name
employees --zip-file
fileb://build.zip --publish",
    "clean": "rm build.zip",
    "build": "zip -r build.zip
node_modules index.js app.js
```

```
routes.js"
```

If you have successfully configured the aws cli, executing the following command:

```
1. npm run deploy
```

Will:

1. Clean the old build
2. Create a new one by packing/zip the required files
3. Publish it to AWS Lambda

When the deploy is completed, you should receive a JSON response with details about the version of the lambda and some other things. In order to be sure it's successfully deployed, you can go to the Amazon Console -> AWS Lambda and check when was the last update of the lambda. If it was a minutes ago, congrats! You are now able to deploy your application to **AWS** with one single command 🙂

The API endpoints are now available and you can test them:

```
1. GET
2. {apiUrl}/employees - Return all
employees
3. {apiUrl}/employees/{employeeId} -
Return specific employee
4.
5. POST
```

```
{apiUrl}/employees - Add Employee
{
  "name": "Test"
}

PUT
{apiUrl}/employees - Update/Edit employee
{
  "id": "ee344452-7f22-4abf-99e6-9b5be668b4f5", // employee id
  "name": "Test"
}

DELETE
{apiUrl}/employees/{employeeId} - Delete specific employee
```

Take your time and test it. I hope everything is working for your as it works for me 🙂

Implement local development capabilities using Docker Compose (for easier development and testing)

Now comes the question, how we can develop and test the things locally before deploy. Something very important in order to avoid bad code in the so called production.

A prerequisite for this job is to install **Docker Compose** on your local machine. I will not dive

deep in it as there is a plenty of information over the internet.

When you are done with the installation, create a file **docker-compose.yml** and fill it with the following content:

```
version: '2'
services:
  dynamodb:
    container_name: dynamodb
    image: 'amazon/dynamodb-local:latest'
    entrypoint: java
    command: '-jar DynamoDBLocal.jar -sharedDb'
    restart: always
    volumes:
       - 'dynamodb-data:/data'
    ports:
       - '8080:8000'
volumes:
  dynamodb-data:
    external: true
```

This is a configuration file and that's how we tell docker compose to create a **DynamoDB** for us. It's ready to use container solution for us which is easier than installing and configuring it locally.

In order to start the DynamoDB instance, we will create one more script in **package.json**

```
 "dynamodb-local-run": "docker-compose up",
```

You can test in by **npm run dynamodb-local-run**. The local instance of **DynamoDB** will be available on port **8080**.

The database is now available and up, but it's empty. We have to create a table, but in order to do that we will need the table model.

We can take the one from our already existing table in AWS, but it will need some tweaks in order to be in the same format as expected by the **aws-cli**. So, you can use the following one:

```
{
    "TableName": "employees",
    "KeySchema": [
      {
        "AttributeName": "id",
        "KeyType": "HASH"
      }
    ],
    "AttributeDefinitions": [
       {
         "AttributeName": "id",
         "AttributeType": "S"
       }
     ],
     "ProvisionedThroughput": {
       "ReadCapacityUnits": 1,
       "WriteCapacityUnits": 1
     }
}
```

Create a new file in your project with name **employee-table-model.json** and paste that model there.

One more script will be needed to create the table. Copy, Paste the following line in **package.json** scripts.

```
1.     "create-database": "aws
dynamodb create-table --cli-input-
json file://employee-table-
model.json --endpoint-url
http://localhost:8080"
```

What we do is to use the aws cli to create the table and specify the endpoint-url to our local DynamoDB instance.

Run the script by **npm run create-database** and the table will be created, which is indicated by the returned **TableDescription** in JSON format.

So, the database is available the table is created. I promise you, only a few more things left.

The next thing is to create a local entry point for the application, because the current one is adjusted to AWS Lambda and is not suitable for local development.

Create **app-local.js** file in the root folder of your project with the following content:

```
1. const app = require('./app');
2. const port = 3000;
3.
```

```
app.listen(port, () => {
    console.log(`listening on http://localhost:${port}`);
});
```

It's using the already available app logic and the only thing on top of it is to start local server using the listen method provided by express.

One more script will be needed to start the application locally:

```
 "start": "TABLE='employees' node app-local",
```

We are setting the Table environment variable to **employees** and executing the local development file with **node app-local**. If it was successfully started, you should see on the console the following output:

```
listening on http://localhost:3000
```

The routes mentioned and tested earlier should be working now locally.

I hope you liked this article and learned something new 🙂

Good luck!

Viktor Borisov

Viktor Borisov is a full-stack JavaScript Developer and teaching enthusiast. His specialties are vanilla JS, Node.JS, AWS and Angular.

API Gateway | AWS | AWS Lambda | DynamoDB | Express | NODE.JS | Serverless

« Previous Post
Check if Object is empty in Javascript

Next Post »
How to convert string to number in JavaScript (BE CAREFUL!)