Scotch.io is shutting down January 2022. It's been a great ride. Thank you for reading.

Follow our founder @chris__sev for more adventures. (https://twitter.com/chris__sev)

In this post we'll be building a node application in Typescript that will be deployed to a lambda function in Amazon Web Services (AWS). Lambdas are used for a variety of tasks and can be written in popular programming languages like C#, Go, Java, Python, and even PowerShell. AWS offers node in a few versions, and for this post we'll be targeting the latest version which, as of this writing, is 8.10.

There are many good reasons to create a node project in typescript, and there are certainly other options out there, like using babel (https://babeljs.io/), but this post assumes the reader is aware of these other options, has weighed the pros and cons, and is interested in this stack. To give a quick background for my reasons, it's a great option for teams who work regularly on an Angular project and want to transfer their typescript skills and knowledge to the backend with lambdas written in the same language as Angular.

If it's not already installed, go ahead and get the latest node and npm versions installed, which at the time of this post, node 10.13.0 and npm 6.4.1 are available. With that done, let's get started!

Table of Contents

# Project Setup

Create a new directory named **hello-ts-lambda**, switch into that directory, and initialize the node project with `npm init -y`. The **package.json** can be edited to look something like this:

JSON

```json
{
  "name": "hello-ts-lambda",
  "version": "0.1.0",
  "description": "A simple typescript node
  "scripts": {},
  "keywords": ["node", "typescript", "lamb
  "license": "ISC"
}
```

Now to convert it into a typescript project.

# Typescript Configuration

Install typescript globally so it can be referenced in the CLI by running `npm i -g typescript` . With that installed globally, `tsc -v` should output *Version 3.1.6*. Initialize typescript in the project by running: `tsc --init` . This will create the **tsconfig.json** file that will control language features, modules, transpilation, relative pathing, etc... To learn more about these configurations, check out the typescript docs (https://www.typescriptlang.org/docs/handbook/tsconfig-json.html)
. The created file will target es5, commonjs modules, enable strict, and turn on esModuleInteropm plus have other options in the JSON file commented out (which is not valid json, but typescript doesn't seem to care).

Let's modify this config to look like this:

JSON

```json
{
  "compilerOptions": {
    "target": "es2017",
    "module": "commonjs",
    "outDir": "./dist",
    "strict": true,
    "baseUrl": "./",
    "typeRoots": [
      "node_modules/@types"
    ],
    "types": [
      "node"
    ],
    "esModuleInterop": true,
    "inlineSourceMap": true
  }
}
```

Awesome! ...but what does it all mean? Let's look at the key options.

- »

`"target"` : specifies the ecmascript version to target, which determines the language features to support. The node version in lambda we're targeting is 8.10 and it supports most of the es2017 features.

- »

  `"module"` : This specifies the module system to transpile to. In node, that's going to be commonjs.

- »

  `"baseUrl"` : Base directory to resolve non-absolute module names

- »

  `"typeRoots"` : List of directories that contains type definitions. This is for installing npm `@types/*` from the DefinitelyTyped project for providing strong typing (giving us intellisense) to popular npm packages.

- »

  `"types:` : This tells typescript what declaration files to include in compilation, in our case, we want to be able to access **node** features like `process.env` and the like.

Well configured IDEs may be seeing an error in the **tsconfig.json** file at this point. Something to the extent of [**ts**]** Cannot find type definition file for 'node'.** What's going on? Well, the config file has defined **node** as a type in the **types** section but we have no type definition files defined anywhere, so let's install them with `npm i -D @types/node` .

After that's installed, ideally, that error should go away, but some IDEs may require a restart.

# Hello World

Let's get a simple hello world working. Create the directory and file **./src/index.ts** . Now to spit out *"Hello World"* into the console add:

TYPESCRIPT

```typescript
console.log('Hello World');
```

To run it, execute `node src/index.ts` . Wait! How can we run a `node` command on a typescript file!? This is madness! No, this is typescript, a superset of javascript. Any valid `.js` file should be able to be renamed to `.ts` and it still be able to be executed using the `node` command. Let's try adding some strong typing and see what happens.

```typescript
var hw: string = 'Hello World';
console.log(hw);
```

We should see an error like this:

```
(function (exports, require, module, **filenam


SyntaxError: Unexpected token :
    at ...
    ...
```

Node doesn't understand the unexpected colon placement, and even if it did, the `string` keyword would throw it off. At this point we need to transpile by simply running `tsc` . In our **tsconfig.json** file, the **outDir** was defined to output transpilations to the **./dist** directory, which should now be created with an **index.js** file in it. Running `node dist/index.js` will now spit out *Hello World* into our console again, but this is cumbersome to do every time. Enter ts-node.

With that installed globally (`npm i -g ts-node`), run the command `ts-node src/index.ts` It will handle the transpilation on the fly and then execute the command to spit out our *Hello World output* once more.

# Lambda Entrypoint

Now that we have a working node/typescript project, let's configure that **src/index.ts** file as an entry point for a lambda. Looking at the
AWS documentation on a lambda handler written in node
(https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-handler.html)
, it explains how lambda will invoke the code on the handler object.

JAVASCRIPT

```javascript
exports.myHandler = function(event, contex
  // or
  // callback("some error type");
}
```

"If you are using runtime version 8.10, you can include the async keyword:"

JAVASCRIPT

```javascript
exports.myHandler = async function(event,
  ...

  // return information to the caller. }
```

And that's what we're after, so let's update our **src/index.ts** file to look like this:

TYPESCRIPT

```typescript
export const handler = async (event: any =
    console.log('Hello World!');
    const response = JSON.stringify(event,
    return response;
}
```

Run the `tsc` command and let's look at our output **dist/index.js** file:

JAVASCRIPT

```javascript
"use strict";
Object.defineProperty(exports, "__esModule
exports.handler = async (event = {}) => {
    console.log('Hello World!');
    const response = JSON.stringify(event,
    return response;
};//# sourceMappingURL=data:application/js
```

Comparing that to the AWS docs of what it expects, we see the async method signature that we're expecting:

`exports.handler = async (event) => { … };`

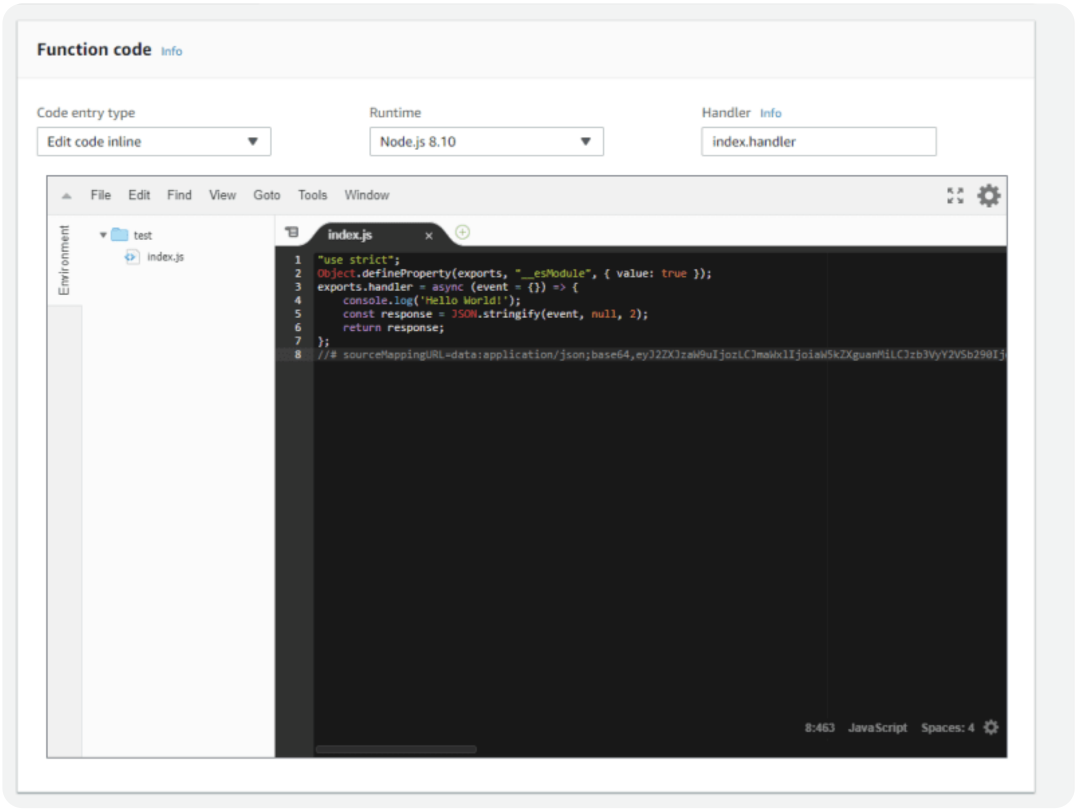
Awesome! Does it really work though? Let's try it out! In the CLI run `ts-node` and follow the along with the commands below:

```
> import { handler } from './src/index';
{}
> handler().then(r => console.log(r));
Hello World!
Promise {
  ...
  }
> {}
> handler({ foo: 'bar' }).then(r => console.lo
Hello World!
Promise {
  ...
  }
> {
  "foo": "bar"
}
```

We imported the handler function and did a simple execution of it. Note that async functions have to return a Promise, so in this example we're accessing the return value within the **then** callback. We could do something like this though:

```
> (async () => {
... const result = await handler({ foo: 'bar'
... console.log(result);
... })();
Hello World!
Promise {
  ...
  }
> {
  "foo": "bar"
}
```

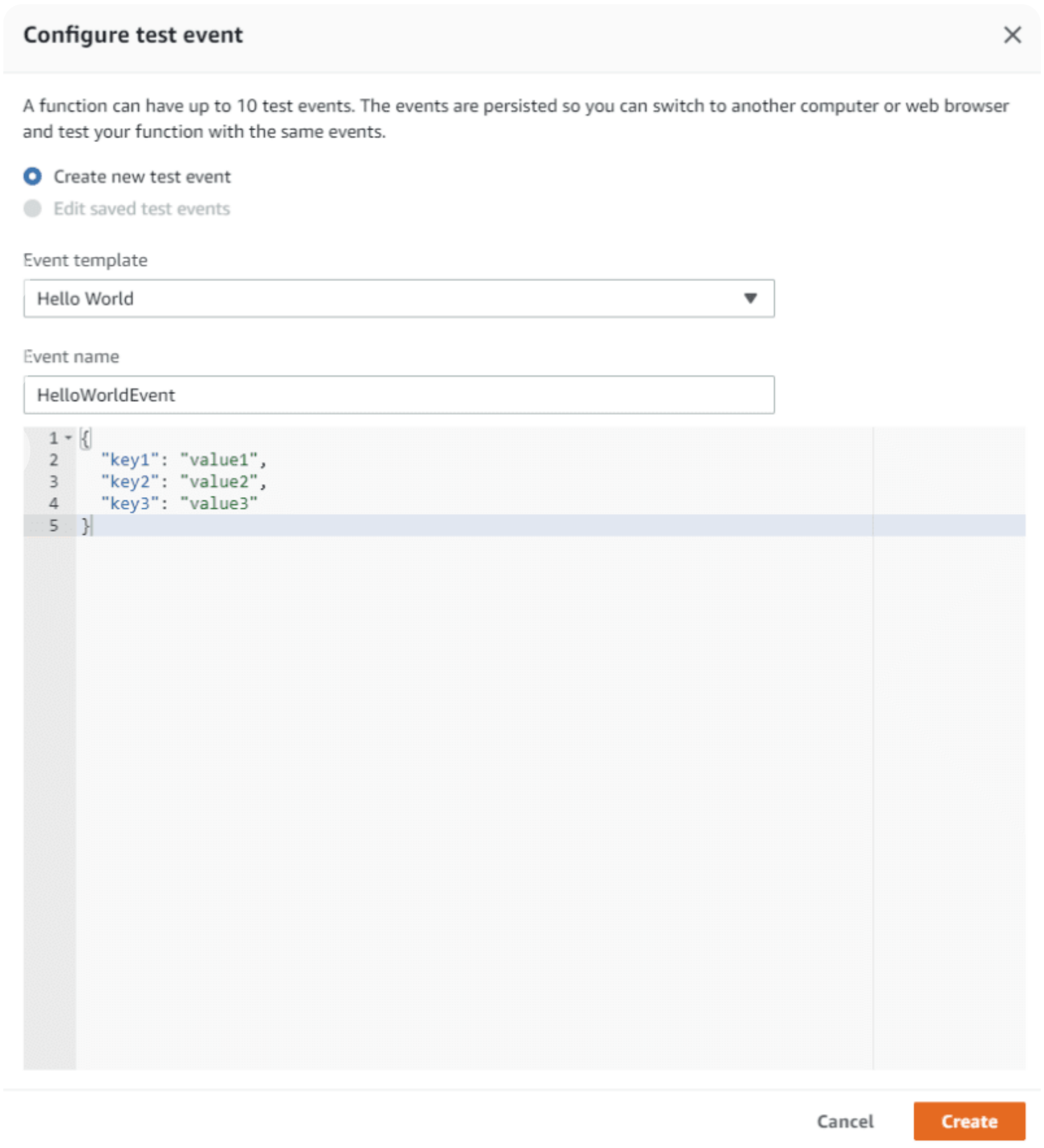
At this point we can see it working locally, now let's zip up the **dist/index.js** file and upload it to a lambda. Looking to the
AWS docs (https://docs.aws.amazon.com/lambda/latest/dg/nodejs-create-deployment-pkg.html)
, there are detailed instructions for how to do this. Here's a screenshot of the uploaded **index.js** file's code:



Code entry type: Edit code inline
Runtime: Node.js 8.10
Handler: index.handler

```
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
exports.handler = async (event = {}) => {
    console.log('Hello World!');
    const response = JSON.stringify(event, null, 2);
    return response;
};
//# sourceMappingURL=data:application/json;base64,eyJ2ZXJzaW9uIjozLCJmaWxlIjoiaW5kZXguanMiLCJzb3VyY2VSb290Ij
```

Create a new test event:



Configure test event

A function can have up to 10 test events. The events are persisted so you can switch to another computer or web browser and test your function with the same events.

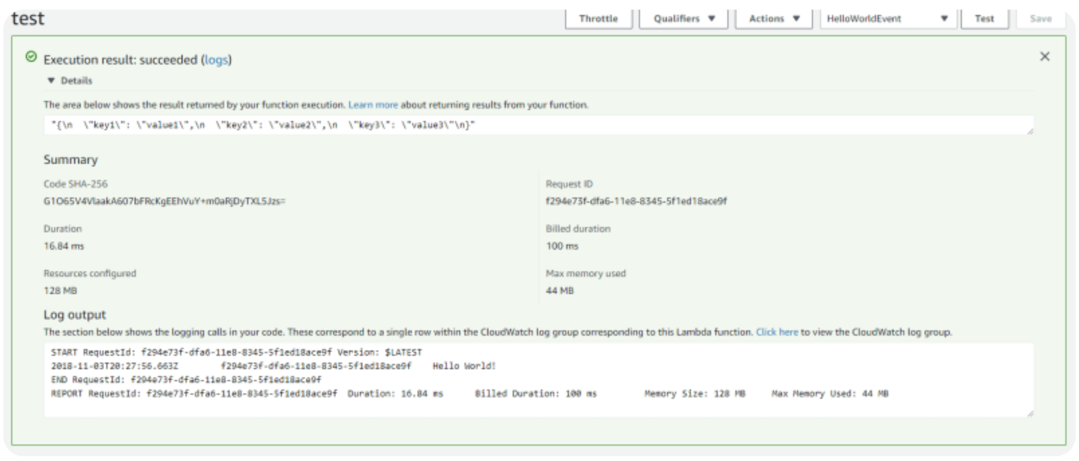
[✓ Create new test event]
Edit saved test events

Event template: Hello World

Event name: HelloWorldEvent

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

and here's the output:



Execution result: succeeded (logs)

Details

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

```
"{\n  \"key1\": \"value1\",\n  \"key2\": \"value2\",\n  \"key3\": \"value3\"\n}"
```

Summary

Code SHA-256	Request ID
G1O65V4VlaakA607bFRcKgEEhVuY+m0aRjDyTXL5Jzs=	f294e73f-dfa6-11e8-8345-5f1ed18ace9f
Duration	Billed duration
16.84 ms	100 ms
Resources configured	Max memory used
128 MB	44 MB

Log output

The section below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.

```
START RequestId: f294e73f-dfa6-11e8-8345-5f1ed18ace9f Version: $LATEST
2018-11-03T20:27:56.663Z	f294e73f-dfa6-11e8-8345-5f1ed18ace9f	Hello World!
END RequestId: f294e73f-dfa6-11e8-8345-5f1ed18ace9f
REPORT RequestId: f294e73f-dfa6-11e8-8345-5f1ed18ace9f	Duration: 16.84 ms	Billed Duration: 100 ms	Memory Size: 128 MB	Max Memory Used: 44 MB
```

# Conclusion

In this post we created a node project from scratch, initialized typescript, wrote a simple hello world file, and uploaded the output to a lambda function. These are the basic building blocks to get this project working. From here the project will need some testing added, code coverage configured, linting, automation setup in scripts for CI/CD, and other finishing touches that I hope to cover in future posts.

Feel free to leave a comment if something wasn't clear or to provide any other kind of feedback.