Getting Started with AWS Lambda

When building software, sometimes you just want to set up a way to perform one simple, self-contained task or action with your code. Going through all the work of creating, setting up, and configuring an entirely new application and the handful of different environments necessary to properly test and deploy your “miniature app” often feels like overkill, almost like serving up a tiny bit of food on a massive plate. The food tastes fine and serves a good purpose by giving you some energy, but you really don’t need that much plate, all that overhead! How can we find the proper size of “plate” for our simple, self-contained task we just coded up? Enter AWS Lambda.

Source: New York Times

AWS Lambda bills itself as a “Function as a Service” or “serverless” solution. AWS’s promo page for Lambda describes it like so: “Run code without thinking about servers. Pay for only the compute time you consume.” At the time of writing this, Lambda allows you to upload C#, Java, Node.js, or Python code for your Lambda function. You can quickly configure things like environment variables needed for the function and the amount of memory to allocate toward the function’s execution, and you can even test your function straight from the browser! When considering solutions for how to host our one-off, self-contained tasks at Experience, all of these things made our choice to use AWS Lambda a no-brainer. We were able to get up and running very quickly with Lambda, and I’m going to show you our real-world use case and how simple it is to go from step 0 to working in production.

At Experience, we partner with several hundred sports and live entertainment properties around the world that use our ticketing solutions, namely our mobile web application. We allow each partner to configure several aspects of the look and feel of the web application to fit their organization’s branding, e.g. colors, logos, and fonts. All of these things that our partners configure need to make their way down to users’ browsers one way or another, and the way we’ve chosen to do this is, naturally, in CSS. If we were to include all of the custom CSS rules for every partner into one massive file, it would end up being around 1 MB gzipped, which is unacceptably large. Instead, we opted to generate one CSS file per partner that would contain our base CSS rules that remain constant across all partners as well as that partner’s bespoke styles. This approach would bring the size of each CSS file down around 70 KB gzipped, which is a massive improvement. Each partner’s version of our web application is already given its own subdomain (e.g. hawks.expapp.com, falcons.expapp.com, etc.) to create clear separation between each partner’s offerings, so it made sense to give each subdomain its own stylesheet.

Our requirements were pretty straightforward. The partner-specific styles are stored in a table in our Postgres database, so we needed a simple way of querying that table and sending that data to our Grunt process, which would then inject the style values into a Sass file and spit out the CSS file for each partner.

Most of our application code is written in the Groovy language, so I decided to write this Lambda function code in Groovy as well. Groovy runs on the JVM, so we can easily compile our code, package it up, and upload that to AWS, and Lambda will never know the difference.

Now onto the code…

```
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.RequestHandler

class GroupStylesGenerator implements RequestHandler<Map, Map> {
   Map handleRequest(Map request, Context context) {
        def dbName = request.nonProdDbName
        def sqlResults = runSqlQuery(dbName)
        return getGroupStylesBySubdomain(sqlResults)
    }

    static def runSqlQuery(dbName) {
        def sql = initConnection(dbName)
        def results = sql.getQueryResults('''
            SELECT ... FROM my_table ...
        ''')
        sql.closeConnection()
        return results
    }

    static def getGroupStylesBySubdomain(sqlResults) {
        ...
    }
...

}
```

Note: This is leaving out some implementation details unnecessary for this blog post, so just know it wasn't *quite* this simple for our complete solution.

This is our implementation of the code that we'll upload to Lambda. Let's go through it from top to bottom. We're importing a couple classes from AWS's Lambda library, which can be found here. We're implementing the RequestHandler class, which is required for any JVM code that will handle an incoming request on Lambda. Our handleRequest method is the method

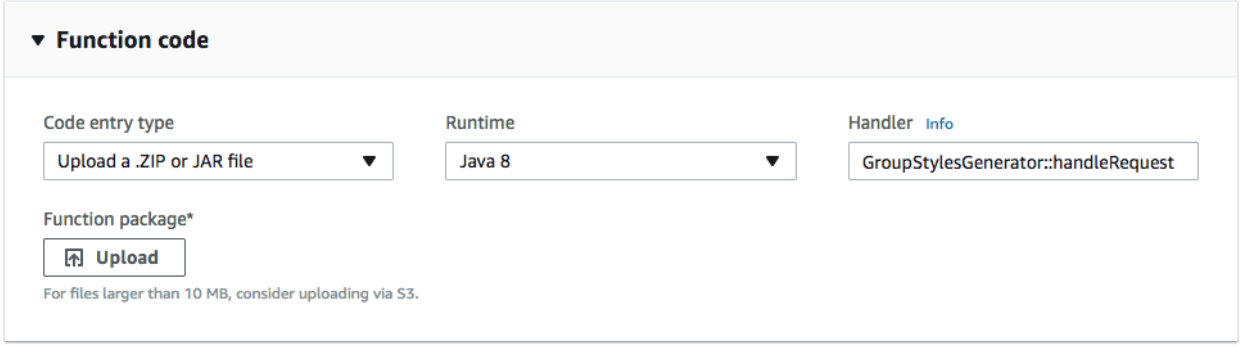
that will be run when we send an HTTP request to the endpoint that is exposed for us by Lambda. There's nothing special about that method name; we'll manually configure our handler method in the Lambda web interface. Our handleRequest method takes a request param and a context param. The request param will contain the request payload, and the context param isn't needed for our implementation. That request param can be a variety of types, such as a Map, a Stream, or a POJO type of your own. For our purposes, we're taking in a JSON payload and also responding in JSON, so Lambda can most easily use the Map type to convert to and from JSON on our behalf. You'll see the interface typing for RequestHandler of <Map, Map>, signifying that we'll be taking in a Map and returning a Map, respectively. Same goes for our method signature; we have a request param type of Map and a return type of Map.

Next, we grab the database name from our request so we can use this script in different environments, run our SQL query, and then run our getGroupStylesBySubdomain method, which essentially organizes the query results and returns a Map. When we return the result of that method from our handleRequest method, Lambda takes that Map, converts it to JSON, and responds to the original request with that JSON data.

After writing our Groovy code, it's time to zip everything up and upload it to Lambda. We just need to include our libraries (the Groovy library, the PostgreSQL library, and the aforementioned Lambda core library) in a lib folder adjacent to our Groovy file. Then we can run a quick Bash script to compile and zip all our files up:

```
groovyc -cp "./lib/*" -d build GroupStylesGenerator.groovy
cp -R lib build/lib
cd build
zip -r GroupStylesGenerator.zip *.class lib
```

Now we can go into the Lambda web interface, upload our zip file, specify our runtime, and specify our handler.



In order to actually use this in our Grunt process, we need to write some JavaScript that will make the request to our new Lambda function. Here's our code:

```
const lambdaParams = {
    InvocationType: "RequestResponse",
    LogType: "None",
    FunctionName = "getGroupStyles",
    Payload: `{ "dbName": "${process.env.DB_NAME}" }`
};

const AWS = require('aws-sdk');
new AWS.Lambda().invoke(lambdaParams, (error, data) => {
    const responseFromLambda = JSON.parse(data.Payload);
    ...
});
```

We are using the aws-sdk Node package, which makes it quite simple to work with AWS in JavaScript. We have our AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables set so that the AWS SDK can authenticate our account and call our lambda function. We then create a Lambda instance and invoke the Lambda function. The params we pass include some things like "getGroupStyles", which is what we named our Lambda function in the web interface, and a small JSON payload, which for some reason needed to be a string instead of a plain old object.

Our callback for handling the Lambda function's response is straightforward. It's an error-first callback, so we'll get our style data back as the second parameter. Lambda wraps a JSON object around the response data and also stringifies the response data, so we needed to do a JSON.parse(data.Payload) to "unwrap" the data we wanted. Now we can use the data we've gotten from our Lambda function and inject it into our Sass and generate all of our CSS files!

Now, when we push our web app to production, our Grunt task sends a single request to our Lambda function, the JVM environment quickly spins up, the Groovy code queries our database, and it returns some JSON for Grunt to handle, and this all happens in just a few seconds. This was a perfect use case for AWS Lambda, and it has been working well for us for a few months now.

If working on this type of thing sounds interesting to you, come work with us!