RUNNING FFMPEG ON AWS LAMBDA FOR 1.9% THE COST OF AWS ELASTIC TRANSCODER

BY EVAN SANGALINE (HTTP://SANGALINE.COM) | MAY 2, 2018

Building a Media Transcoder with Exodus, FFmpeg, and AWS Lambda

When delivering media content over the internet, it's important to keep in mind that factors like network bandwidth, screen resolution, and codec support will vary drastically between different devices and connections. Certain media encodings will be better suited for certain viewers, and transcoding source media to multiple formats is a must in order to ensure that you're delivering the best possible experience to your users. Netflix, for example, encodes each title in at least 120 different formats (https://medium.com/netflix-techblog/complexity-in-the-digital-supply-chain-958384cbd70b) and optimizes the encoding settings on a per-title basis (https://medium.com/netflix-techblog/per-title-encode-optimization-7e99442b62a2). They're obviously a bit of an outlier, but even something as simple as using VP9 (https://en.wikipedia.org/wiki/VP9) with an H.264 (https://en.wikipedia.org/wiki/H.264/MPEG-4_AVC) fallback in your video tags can have a tremendous impact on user experience.

If you're just transcoding a few files here and there for your own site, then you can probably get away with just running things manually on your own laptop. Things get a bit more complicated if you need to trigger transcoding tasks programmatically in a scalable way. For instance, imagine that you would like to build a service for extracting MP3s from video files. There are a number of ways that one could go about accomplishing something like this. You could use Amazon's AWS Elastic Transcoder (https://aws.amazon.com/elastictranscoder) service, or you could run a server with FFmpeg (https://www.ffmpeg.org/) installed and handle the transcoding yourself. We'll be building a media transcoding service in this tutorial, but we won't use either of these two methods.

Instead, we'll be using Exodus (https://github.com/intoli/exodus) to bundle FFmpeg inside of an AWS Lambda (https://aws.amazon.com/lambda) function for performing transcoding tasks. This might sound similar to running FFmpeg on our own server, but there are some significant differences. Using Lambda means that our transcoder will automatically scale to handle whatever volume of requests we throw at it, and that our costs will be proportional to usage. In those ways, our Lambda transcoder is more similar to using Elastic Transcoder than to managing our own infrastructure.

So why not just use Elastic Transcoder instead of Lambda? Well, for one thing, this this is a Lambda tutorial! The techniques that we'll cover for including complex native binaries in Lambda functions will be generally useful outside the context of transcoding, and that's the primary purpose of this article. That said, the cost savings of using Lambda rather than a more specialized service can be massive; so there is some practicality to the choice as well.

Let's do a quick price comparison of Elastic Transcoder and the transcoder Lambda function that we'll build. The cost of transcoding audio on Elastic Transcoder is $0.0045 per minute of audio. The Lambda pricing is a bit more complicated because it depends on both the execution time and the RAM allocation. The cost per GB-second is $0.00001667, and transcoding a minute of audio with the function that we'll develop requires just shy of 5 GB-seconds. This puts the overall cost at $0.00008273 per minute of audio, a full factor of 54 times less than Elastic Transcoder.

On top of that, the perpetually-free tier of Elastic Transcoder allows for only 20 minutes of audio transcoding per month. Lambda's perpetually-free tier is 400k GB-seconds per month, which translates to *56 days* of free audio transcoding per month. A comparable amount of transcoding would cost $362 with Elastic Transcoder. These free tiers can be really important for small side projects, one of the areas where Lambda really shines.

I also want to stress that the Lambda function we'll be building is not anything near a full drop in replacement for Elastic Transcoder. It's designed for audio-only transcoding, it can only support a maximum of about 8 minutes of audio per invocation, it has no error handling, *etc.* The point of the comparison isn't that Elastic Transcoder is a ripoff, it's that Lambda can often result in significant cost savings if you have a well-defined use case that's compatible with the limitations that Lambda imposes. Lambda is a relatively good fit for something like a simple Youtube MP3 downloader service, but that's already pushing the limits of what a single Lambda invocation can handle.

This tutorial will walk through the full process of developing and deploying a Lambda function for transcoding audio, but it can also often be helpful to see how everything fits together in one place. All of the finished code for the project is also available in the intoli-article-materials (https://github.com/intoli/intoli-article-materials/tree/master/articles/youtube-mp3-downloader) repository, so feel free to head over there if you would like to skip ahead and see the finished product. We post supplementary materials for all of our technical articles there, so starring that repository is also a good way to find out when we're working on something new and interesting! With that said, let's get started on actually building our microservice!

Bundling FFmpeg for Lambda

One of the most challenging components of developing complex Lambda functions is packaging any native dependencies that a function relies on. If you've ever worked with Docker (https://www.docker.com/), then you probably know how convenient it is to be able to create your own container images with preinstalled requirements. Even though Lambda functions run in containers, there's no comparable way to customize the installed packages on the system. You get to put 250 MB of files in `/var/task` , and *that's it*.

The reason that packing native code can be such a challenge is that Linux software tends to be dynamically rather than statically linked. This is a necessity for allowing different packages to reuse the same dependencies, but it also makes relocating software extremely challenging. Let's take a quick look at FFmpeg as an example.

The `ffmpeg` binary on my computer is only a measly 260 KB. If you were to include this file as part of your Lambda function and try to run it, you would get the following error.

```
./ffmpeg: error while loading shared libraries: libavdevice.so.57: cannot open shared object file: No such file c
```

We can use ldd (http://man7.org/linux/man-pages/man1/ldd.1.html) to list all of the shared dependencies that FFmpeg needs in order to run. Executing `ldd $(which ffmpeg)` reveals a staggering 104 direct dependencies, and that's not even counting the secondary dependencies that these libraries themselves depend on. Here are the first ten of them to give you the general idea.

```
        libavdevice.so.57 => /usr/lib/libavdevice.so.57 (0x00007ff8ad94e000)
        libavfilter.so.6 => /usr/lib/libavfilter.so.6 (0x00007ff8ad4a7000)
        libavformat.so.57 => /usr/lib/libavformat.so.57 (0x00007ff8ad05b000)
        libavcodec.so.57 => /usr/lib/libavcodec.so.57 (0x00007ff8ab98c000)
        libavresample.so.3 => /usr/lib/libavresample.so.3 (0x00007ff8ab76c000)
        libpostproc.so.54 => /usr/lib/libpostproc.so.54 (0x00007ff8ab54e000)
        libswresample.so.2 => /usr/lib/libswresample.so.2 (0x00007ff8ab331000)
        libswscale.so.4 => /usr/lib/libswscale.so.4 (0x00007ff8ab0a9000)
        libavutil.so.55 => /usr/lib/libavutil.so.55 (0x00007ff8aae23000)
        libm.so.6 => /usr/lib/libm.so.6 (0x00007ff8aaad7000)
```

It's pretty clear that the missing `libavdevice.so.57` file is really just the tip of the iceberg here.

So say that you tracked down *all* of these dependencies, placed them in a `lib/` subdirectory, and then set the `LD_LIBRARY_PATH` environment variable to `/var/task/lib` before invoking FFmpeg in your Lambda function. Would everything work then? Nope! Well… maybe, but probably not.

Even if all of the correct libraries are in the library search path, `ffmpeg` –and all ELF (https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) executables, for that matter–will have a hardcoded interpreter path that the kernel will use to start the program. This can be checked by printing out the ELF program headers with `readelf --program-headers $(which ffmpeg)` (on my system it's, `/lib64/ld-linux-x86-64.so.2`). If that file doesn't exist in the Lambda execution environment, then FFmpeg won't run–even if all of the libraries are the search path. What's even worse is that if the interpreter *is there*, but was compiled with a different glibc (https://www.gnu.org/software/libc/) version, then you're likely to get either glibc relocation errors or extremely subtle buggy execution behavior.

The official AWS recommendation (https://aws.amazon.com/blogs/compute/running-executables-in-aws-lambda/) for dealing with these issues is to compile your own native binaries in the EC2 AMI that Lambda is based on. A problem with that is that the system library versions in Lambda are somewhat ancient, and incompatible with a lot of newer software. This can quickly turn into a rabbit hole where you compile library after library, only to eventually discover that it's absolutely never going to work due to these incompatibilities.

IF YOU TRY TO COMPILE GTK+ 3 ON AMAZON LINUX
INSTRUCTO
YOU'RE GONNA HAVE A BAD TME

If chasing wild geese isn't really your cup of tea, then I'm glad to tell you that we've built a tool called Exodus (https://github.com/intoli/exodus) which greatly simplifies the process of relocating native binaries from one Linux system to another. The process of bundling a local copy of FFmpeg to run on Lambda, is literally just this.

```
# Install the `exodus_bundler` package, if you haven't already.
pip install --user exodus_bundler
export PATH="${HOME}/.local/bin/:${PATH}"

# Create an `ffmpeg` bundle and extract it in the current directory.
exodus --tarball ffmpeg | tar -zx
```

That command will create an `exodus/` subdirectory which includes a fully relocatable copy of FFmpeg that you can run on Lambda or any other Linux system. You can then run `./exodus/bin/ffmpeg` and all of library, linker, environment variable, *etc.* stuff will be dealt with automatically. The bundle itself weighs in at only 110 MB, allowing it to fit comfortably within the 250 MB limit for Lambda functions. In contrast, a Nix (https://nixos.org/nix/) installation of FFmpeg occupies 367 MB because it includes many non-essential dependencies.

I better stop myself from going into too much detail about how this works, but you can check out the Exodus README (https://github.com/intoli/exodus) and What's New in Exodus 2.0 (/blog/exodus-2/) for a lot more details. It's actually really cool, it's just *slightly* tangential to the topic of building a YouTube MP3 downloader.

Setting Up S3 and Access Permissions

Now that we have FFmpeg bundled up, it's time to start configuring AWS. All of the instructions here will use the AWS Command-Line Interface (https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html), and it's assumed that you've already configured your credentials and set a default region (https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html). It's also generally assumed that you have some basic familiarity with AWS API Gateway (https://aws.amazon.com/api-gateway/), AWS Identity and Access Management (IAM) (https://aws.amazon.com/iam/), AWS Lambda (https://aws.amazon.com/lambda/), and AWS S3 (https://aws.amazon.com/s3/). You can probably still make it through the guide if you haven't used one of these specific services before, but this tutorial really isn't meant to be a first introduction to what Amazon Web Services are.

Something else to keep in mind here as we start configuring things is that this tutorial is the first part of a two part series. The second part of the series will focus on building YouTube MP3 downloader service, while this part is solely focused on bundling FFmpeg and making a self-contained transcoding function. The transcoder instructions should be able to stand on their own, but we'll be naming things and setting permissions in anticipation of these resources will later be used to build a YouTube MP3 downloader service.

When our transcoding Lambda function produces output files, it will need a place to store them. S3 is the natural choice here, so we'll start by creating a new bucket using aws s3 mb (https://docs.aws.amazon.com/cli/latest/reference/s3/mb.html) command. Note that S3 bucket names need to be globally unique, so you'll need to change `youtube-mp3-downloader` to something else here.

```
# Store this for later use.
export bucket_name="youtube-mp3-downloader"

# Actually create the bucket.
aws s3 mb "s3://${bucket_name}"
```

If all goes well, then the command should echo back a success message like this.

```
make_bucket: youtube-mp3-downloader
```

Now we'll need to set up a role for our Lambda function, and grant it the necessary permissions. We're eventually going to expose our YouTube MP3 downloader Lambda function via API Gateway, so we'll list both API Gateway and Lambda as the principal services in our role policy document. If you're only interested in the transcoding Lambda function, and not the whole YouTube MP3 downloader, then you can leave off the API Gateway principal. JSON can be pretty cumbersome to write on a single line, so we'll store our multi-line policy in an environment variable called `role_policy_document` using a here document (https://www.tldp.org/LDP/abs/html/here-docs.html).

```
read -r -d '' role_policy_document <<'EOF'
  {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
          "Service": [
            "apigateway.amazonaws.com",
            "lambda.amazonaws.com"
          ]
        },
        "Action": "sts:AssumeRole"
      }
    ]
  }
EOF
```

Now we'll use this policy document to create a new role using the aws iam create-role (https://docs.aws.amazon.com/cli/latest/reference/iam/create-role.html) command. You'll also notice that I'm storing the response in an environment variable here, and then using jq (https://stedolan.github.io/jq/) to extract a particular value from the JSON response (in this case, the role's Amazon Resource Name/ARN (https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html)). This is just a convenience so that you can copy and paste future commands without needing to manually replace things like ARNs, the bucket name, function names, *etc.* We'll follow this pattern for any future AWS commands where we'll want to store parts of the response in variables.

```
# Store this for later use.
export role_name="YoutubeMp3DownloaderRole"

# Create a new role and store the JSON response in a variable.
response="$(aws iam create-role \
    --role-name "${role_name}" \
    --assume-role-policy-document "${role_policy_document}")"

# Echo the response in the terminal.
echo "${response}"

# Store the role ARN for future usage.
role_arn="$(jq -r .Role.Arn <<< "${response}")"
```

This should store the ARN for the new resource in the `role_arn` environment variable, and echo out some details about the newly created role.

```
{
    "Role": {
        "Path": "/",
        "RoleName": "YoutubeMp3DownloaderRole",
        "RoleId": "AROAIYOVCJF3FXJNTQ2II",
        "Arn": "arn:aws:iam::421311779261:role/YoutubeMp3DownloaderRole",
        "CreateDate": "2018-04-03T19:15:45.606Z",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": [
                            "apigateway.amazonaws.com",
                            "lambda.amazonaws.com"
                        ]
                    },
                    "Action": "sts:AssumeRole"
                }
            ]
        }
    }
}
```

After creating our role, we'll need to give it adequate permissions. The following policy will grant full access to the S3 bucket that we created earlier, as well as full access to API Gateway and permission to invoke and Lambda function. In production, you would want to specify API Gateway and Lambda function ARNs here, but we don't know what those are here because we haven't created those resources yet! We'll just leave them as wildcards for now, and you can circle back and tighten things up later if you so desire.

```
read -r -d '' policy_document <<EOF
  {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "apigateway:*"
        ],
        "Resource": "arn:aws:apigateway:*::/*"
      },
      {
        "Effect": "Allow",
        "Action": [
          "execute-api:Invoke"
        ],
        "Resource": "arn:aws:execute-api:*:*:*"
      },
      {
        "Effect": "Allow",
        "Action": [
            "lambda:*"
        ],
        "Resource": "*"
      },
      {
        "Effect": "Allow",
        "Action": "s3:ListAllMyBuckets",
        "Resource": "arn:aws:s3:::*"
      },
      {
        "Effect": "Allow",
        "Action": "s3:*",
        "Resource": [
          "arn:aws:s3:::${bucket_name}",
          "arn:aws:s3:::${bucket_name}/*"
        ]
      }
    ]
  }
EOF
```

Finally, we can apply this policy to the role we created using the aws iam put-role-policy (https://docs.aws.amazon.com/cli/latest/reference/iam/put-role-policy.html) command.

```
# Store this for later use.
export policy_name="YoutubeMp3DownloaderPolicy"

# Apply the policy.
aws iam put-role-policy \
    --role-name "${role_name}" \
    --policy-name "${policy_name}" \
    --policy-document "${policy_document}"
```

At this point, we have a place to store our files and we have a role with a policy in place that will allow us to build the full YouTube MP3 downloader backend. The permissions stuff is less fun than actually writing and deploying code, but it's crucial in order for our Lambda functions and API to work. Now that it's out of the way, we can move on to the more exciting parts!

The Transcoder Lambda Function

We'll be writing our Lambda functions using Node (https://nodejs.org/en/), targeting the nodejs v6.10 AWS Lambda runtime, and using Yarn (https://yarnpkg.com/en/) to manage our dependencies. Amazon actually also just announced a new nodejs v8.10 AWS Lambda runtime (https://aws.amazon.com/blogs/compute/node-js-8-10-runtime-now-available-in-aws-lambda/). I would generally recommend using the most recent target for any new projects, but I had already written the following code to target v6.10 before they announced the new runtime. This shouldn't really make too much of a difference, just know that you're now free to use `async` / `await` syntax instead of promise chains if you use the newer target.

Our transcoding function will have only three Node dependencies: the aws-sdk (https://www.npmjs.com/package/aws-sdk) for uploading files to S3, request (https://www.npmjs.com/package/request) for downloading files, and tempy (https://www.npmjs.com/package/tempy) for generating temporary file paths. We can install these by running

```
yarn add aws-sdk request tempy
```

which will place the necessary packages in the `node_modules` directory. Note that we don't *technically* need to install the `aws-sdk` because it will be available by default in the Lambda environment, but it will be convenient to have a local copy during development.

Now, before we start coding, let's lay out how this Lambda function will operate. The function will accept an `event` object which contains the following keys.

- `filename` - The filename to use in the MP3 file's Content-Disposition header (https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Disposition) when a user downloads it. This determines the filename that will be suggested to the user when they save it to their computer.
- `logKey` - An S3 key where the output of FFmpeg will be placed for logging purposes.
- `mp3Key` - An S3 key where the converted MP3 file will be placed.
- `s3Bucket` - The S3 bucket where the log and MP3 files will be placed.
- `url` - The URL where the input audio/video file can be downloaded from.

The function will then follow the following steps.

1. Create temporary filenames to store both the input file and the output MP3.
2. Download the input file from `url` into its temporary location.
3. Invoke the local copy of FFmpeg that we bundled earlier with Exodus, and pass it arguments to transcode the input file to an MP3.
4. Upload the MP3 file to S3 using the `s3Bucket` and `mp3Key` parameters.
5. Upload the output of FFmpeg to S3 using the `s3Bucket` and `logKey` parameters.
6. Delete the temporary files in case the container is reused during a future invocation.

Overall, the logic is actually pretty simple. I think that most of the complexity in putting together a function like this really lies in bundling FFmpeg, and–as we saw earlier–Exodus is extremely helpful there.

To put this all together into an actual Lambda handler, you can add the following code to a file called `transcoder.js` in the same directory where you ran Exodus and Yarn earlier.

```
const child_process = require('child_process');
const fs = require('fs');
const path = require('path');

const AWS = require('aws-sdk');
const request = require('request');
const tempy = require('tempy');

const s3 = new AWS.S3();


exports.handler = (event, context, callback) => {
  // We're going to do the transcoding asynchronously, so we callback immediately.
  callback();

  // Extract the event parameters.
  const { mp3Key, url } = event;
  const filename = event.filename || path.basename(mp3Key);
  const logKey = event.logKey || `${mp3Key}.log`;
  const s3Bucket = event.s3Bucket || 'youtube-mp3-downloader';

  // Create temporary input/output filenames that we can clean up afterwards.
  const inputFilename = tempy.file();
  const mp3Filename = tempy.file({ extension: 'mp3' });

  // Download the source file.
  Promise.resolve().then(() => new Promise((resolve, revoke) => {
    const writeStream = fs.createWriteStream(inputFilename);
    writeStream.on('finish', resolve);
    writeStream.on('error', revoke);
    request(url).pipe(writeStream);
  }))
  // Perform the actual transcoding.
  .then(() => {
    // Use the Exodus ffmpeg bundled executable.
    const ffmpeg = path.resolve(__dirname, 'exodus', 'bin', 'ffmpeg');

    // Convert the FLV file to an MP3 file using ffmpeg.
    const ffmpegArgs = [
      '-i', inputFilename,
      '-vn', // Disable the video stream in the output.
      '-acodec', 'libmp3lame', // Use Lame for the mp3 encoding.
      '-ac', '2', // Set 2 audio channels.
      '-q:a', '6', // Set the quality to be roughly 128 kb/s.
      mp3Filename,
    ];
    const process = child_process.spawnSync(ffmpeg, ffmpegArgs);
    return process.stdout.toString() + process.stderr.toString();
  })
  // Upload the generated MP3 to S3.
  .then(logContent => new Promise((resolve, revoke) => {
    s3.putObject({
      Body: fs.createReadStream(mp3Filename),
      Bucket: s3Bucket,
      Key: mp3Key,
      ContentDisposition: `attachment; filename="${filename.replace('"', '\'')}"`,
      ContentType: 'audio/mpeg',
    }, (error) => {
      if (error) {
        revoke(error);
      } else {
        // Update a log of the FFmpeg output.
        const logFilename = path.basename(logKey);
        s3.putObject({
          Body: logContent,
          Bucket: s3Bucket,
          ContentType: 'text/plain',
          ContentDisposition: `inline; filename="${logFilename.replace('"', '\'')}"`,
          Key: logKey,
        }, resolve);
      }
    })
  }))
```

```
    .catch(console.error)
    // Delete the temporary files.
    .then(() => {
      [inputFilename, mp3Filename].forEach((filename) => {
        if (fs.existsSync(filename)) {
          fs.unlinkSync(filename);
        }
      });
    });
};
```

You can see here that the transcoding options are all determined by the arguments that we pass to the FFmpeg child process. They're only hardcoded to produce MP3 files here because this function is designed to be used with the YouTube MP3 downloader. You could easily adapt the code to use a different output format, or even to accept codec details as parameters in the invocation event.

Now we'll need to package our Lambda function into a ZIP file in order to deploy our code. An annoying thing about the AWS command-line tools is that they often timeout when uploading large ZIP files to Lambda. The workaround for this is to first upload the ZIP file to S3, and then specify the S3 bucket and key rather than a local file path. The only problem is that we can only do this when we *update* a function's code, we can't do it when we create a new one.

To make sure that the initial function creation succeeds, we'll initially create a minimal ZIP file that doesn't contain any of our dependencies. This won't actually run without the dependencies, but the file will upload quickly and be considered valid by Lambda.

```
# This won't run without the dependencies, but it will upload quickly.
zip youtube-mp3-transcoder.zip transcoder.js
```

We can then use aws lambda create-function (https://docs.aws.amazon.com/cli/latest/reference/lambda/create-function.html) to actually create our Lambda function. One thing to note here is that we're specifying the timeout to be 300 seconds. This is the maximum timeout for a Lambda function, and it's essentially what determines the maximum video length that our transcoder will be able to handle. If something takes longer than 5 minutes to download, transcode, and upload, then our function will fail silently. The maximum video length works out to be around 8 minutes in practice when converting YouTube videos.

```
# Store the function name for later.
export transcoder_function_name="YoutubeMp3TranscoderFunction"

# Create the transcoder function.
aws lambda create-function \
    --function-name "${transcoder_function_name}" \
    --zip-file fileb://youtube-mp3-transcoder.zip \
    --handler transcoder.handler \
    --runtime nodejs6.10 \
    --timeout 300 \
    --role "${role_arn}"
```

This should echo out some additional details about the function that we just created.

```
{
    "FunctionName": "YoutubeMp3TranscoderFunction",
    "FunctionArn": "arn:aws:lambda:us-east-2:421311779261:function:YoutubeMp3TranscoderFunction",
    "Runtime": "nodejs6.10",
    "Role": "arn:aws:iam::421311779261:role/YoutubeMp3DownloaderRole",
    "Handler": "transcoder.handler",
    "CodeSize": 1285,
    "Description": "",
    "Timeout": 300,
    "MemorySize": 128,
    "LastModified": "2018-04-03T19:18:39.769+0000",
    "CodeSha256": "bza/tr23iCeSXXwF+jTm6LJwZzuNFGK76AhNatVGSkQ=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "8e0b54ab-d4cc-43c3-882b-9da73978f440"
}
```

You can see here that the memory size allocated for the function is 128 MB. This is the default value for the function, but it turns out to be just about right for these transcoding tasks; they tend to cap out at around 115-120 MB of peak usage.

Now that we've created our function, we can use the S3 workaround to include our dependencies in the function. The dependencies that we'll need to add to the ZIP file are the packages in `node_modules/` and the FFmpeg bundle in `exodus/`. We can upload our full function bundle to S3 using aws s3 cp (https://docs.aws.amazon.com/cli/latest/reference/s3/cp.html), and then update the function using aws lambda update-function-code (https://docs.aws.amazon.com/cli/latest/reference/lambda/update-function-code.html).

```
# Create a ZIP file with all of the dependencies.
rm -f youtube-mp3-transcoder.zip
zip --symlinks --recurse-paths youtube-mp3-transcoder.zip \
    transcoder.js package.json node_modules/ exodus-2/

# Upload the ZIP file to S3.
aws s3 cp youtube-mp3-transcoder.zip "s3://${bucket_name}/"

# Update the function using the ZIP file on S3 as the code source.
aws lambda update-function-code \
    --function-name "${transcoder_function_name}" \
    --s3-bucket "${bucket_name}" \
    --s3-key youtube-mp3-transcoder.zip
```

That will take a little while to finish because of the file upload, but our transcoding function should be good to go after it does! Let's test it out by using an input file on the ever-so-charming WavSource.com (http://www.wavsource.com/) (I love their motto: "If you've ever said, 'I Love Wavs!' then you've come to the right place!"). We can use any download URL of a WAV on the site to construct an event object to pass to our Lambda function. I chose a Humphrey Bogart quite from The Caine Mutiny (http://www.imdb.com/title/tt0046816/), and constructed the following event.

```
read -r -d '' transcoder_event <<EOF
  {
    "logKey": "fly-right.log",
    "mp3Key": "fly-right.mp3",
    "s3Bucket": "${bucket_name}",
    "url": "http://www.wavsource.com/snds_2018-01-14_3453803176249356/movie_stars/bogart/fly_right.wav"
  }
EOF
```

We can then use the aws lambda invoke (https://docs.aws.amazon.com/cli/latest/reference/lambda/invoke.html) command to invoke our transcoder function with this test event.

```
# Invoke our transcoder function with this test event.
aws lambda invoke \
  --function-name "${transcoder_function_name}" \
  --payload "${transcoder_event}" \
  /dev/null
```

Even though our Lambda function is designed to be asynchronous, the default invocation type of `RequestResponse` will actually wait for the Lambda function to completely terminate before returning. That means that our converted MP3 and the corresponding log file should both be available as soon as this command finishes running. To check that, you can run the following commands to download these files from S3.

```
# Download and play the converted MP3.
aws s3 cp "s3://${bucket_name}/fly-right.mp3" ./
mplayer fly-right.mp3

# Display the log output.
aws s3 cp "s3://${bucket_name}/fly-right.log" -
```

The log output here will look something like this.

```
ffmpeg version 3.4.2 Copyright (c) 2000-2018 the FFmpeg developers
  built with gcc 7.3.0 (GCC)
  configuration: --prefix=/usr --disable-debug --disable-static --disable-stripping --enable-avisynth --enable-av
  libavutil      55. 78.100 / 55. 78.100
  libavcodec     57.107.100 / 57.107.100
  libavformat    57. 83.100 / 57. 83.100
  libavdevice    57. 10.100 / 57. 10.100
  libavfilter     6.107.100 /  6.107.100
  libavresample   3.  7.  0 /  3.  7.  0
  libswscale      4.  8.100 /  4.  8.100
  libswresample   2.  9.100 /  2.  9.100
  libpostproc    54.  7.100 / 54.  7.100
Invalid return value 0 for stream protocol
    Last message repeated 2 times
Guessed Channel Layout for Input Stream #0.0 : mono
Input #0, wav, from '/tmp/dcf018284107303d702c7cbbdcdf663e':
  Duration: 00:00:02.26, bitrate: 88 kb/s
    Stream #0:0: Audio: pcm_u8 ([1][0][0][0] / 0x0001), 11025 Hz, mono, u8, 88 kb/s
Stream mapping:
  Stream #0:0 -> #0:0 (pcm_u8 (native) -> mp3 (libmp3lame))
Press [q] to stop, [?] for help
Output #0, mp3, to '/tmp/ad0f4181c199a247c84cf8b560720a63.mp3':
  Metadata:
    TSSE            : Lavf57.83.100
    Stream #0:0: Audio: mp3 (libmp3lame), 11025 Hz, stereo, s16p
    Metadata:
      encoder         : Lavc57.107.100 libmp3lame
Invalid return value 0 for stream protocol
size=       9kB time=00:00:02.30 bitrate=  31.5kbits/s speed=8.82x
video:0kB audio:9kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 2.874020%
```

This is a pretty typical FFmpeg output; we can see version and configuration information for the FFmpeg build that we're using, the formats and filenames of the input and output files, and some details about the encoding process. Probably the most interesting thing to note here is the speed of `8.82x`, which gives us some idea of the maximum length of audio that we could encode in a single Lambda invocation. If we were to ignore the time for downloading and uploading, then the maximum audio length that we could encode to MP3 would be about 8.82 times 5 minutes, so 44.1 minutes. That's something like an absolute upper bound on what this function would be able to do; the download/upload times subtract from this, as would any decoding that needs to take place for source files that aren't WAV files.

Intermission/Conclusion

So far, we've covered the process of bundling FFmpeg with Exodus and building a basic media transcoding function on AWS Lambda. We've also done some back-of-the-envelope calculations to show that this can be an extremely cost effective alternative to AWS Elastic Transcoder when dealing with transcoding tasks that can be processed within the 5 minute limit for Lambda function invocations. In the next part of this series, Making a YouTube MP3 Downloader with Exodus, FFmpeg, and AWS Lambda (/blog/youtube-mp3-downloader), we'll jump into building a practical YouTube MP3 downloader service which builds upon the transcoding function that we've developed here.

If you're interested in checking out more great content from Intoli, then feel free to browse through other articles on our blog (/blog/) or to subscribe to our monthly article newsletter (http://intoli.us11.list-manage.com/subscribe?u=95178aac513dc319bf87a9c3&id=bb722b0e5d). You can also star or follow our intoli-article-materials repository (https://github.com/Intoli/intoli-article-materials/) on GitHub to find out about new articles, often even before they are published!

SUGGESTED ARTICLES

If you enjoyed this article, then you might also enjoy these related ones.