Ruby on AWS Lambda

NP-Complete

まえがき

ついについについに **AWS Lambda で Ruby** が使えるようになりました。最高 of 最高です。最近ますます価値を高めているサーバーレス技術をついに Ruby でやれるようになったのです。もう JavaScript なんて書かなくて良いぞおおおおおおおおおおお!!!! やったぞおおおおおおおおおおおおお!!!! というわけで今回は AWS Lambda で Ruby を動かしてみるという内容の本です。

この本は、CC0[*1]でライセンスされ、自由に利用できます。コードは np-complete/TechBookFes06[*2]にあります。PDF[*3]でダウンロードもできます。ウィッシュリストはこちらです[*4]。

[*1] https://creativecommons.org/choose/zero/
[*2] https://github.com/np-complete/TechBookFes06
[*3] https://s3.amazonaws.com/np-complete-books/pdf/TechBookFes06.pdf
[*4] https://twishli.st/masarakki

第 1 章

AWS Lambda について

AWS Lambda は **FaaS (Function as a service)** と呼ばれるサービスで、その名の通り、関数を実行するというシンプルな機能をサービス化したものです。

もちろん、その関数はどこかのコンピュータで動いている **はず** なのですが、開発者はそれを意識する必要はありません[*1]。AWS のマネージドサーバで実行されるので、コンピュータを管理する必要もないし勝手にスケールもしてくれます。関数は様々な方法で呼び出すことができ、直接実行するだけでなく、AWS の他のサービスから実行されるようにトリガーを設定することもできます。

メジャーな使い方としては大きく 2 つあり、

- 他の AWS サービスからイベントを受けバッチ処理などを非同期で行う
- ALB や API Gateway が受けたリクエストをリアルタイム処理する

に分けられます。特に後半の使い方がサーバレステクノロジーと言われて最近人気です。

Lamda でのプログラミングは、既存のプログラミング手法とは思想を変える必要があります。ひとつの関数は、UNIX 哲学にもある **ひとつのことを上手くやる** ような作り方をします。Rails アプリのような **モノリシック** な巨大な機能群のかたまりは Lambda の思想とマッチしません。環境はリセットされ、毎回ロードされるので、そもそも大きなライブラリを読み込むことは間違ったやり方です。当然、状態を保存するようなこともできないので、関数の **副作用** にあたるものは、AWS の様々なサービスを組み合わせて実現する必要があります。

1.1 画像アップローダを作ってみよう

ここで例として、Web アプリケーションでありがちな、「画像をアップロードしてついでにサムネイルを作る」という機能を考えてみましょう。

既存のアプリケーションでは、

- ファイルアップロードのリクエストを受け付ける
- アップロードされた画像を S3 に保存する
- サムネイルを作って S3 に保存する
- S3 の URL をデータベースに保存する

[*1] そんなわけないだろ!!!!

- レスポンスを返す

というような 1 つのエンドポイントを作ることが多いでしょう。言語やフレームワークによっては、何も考えなくても全てやってくれるライブラリ[*2]が存在することもあるでしょう。

これを、AWS の機能と Lambda を使うと、このように作れます。

- Cognito で一時的な権限を発行する
- ブラウザから直接 S3 に画像をアップロードさせる
- S3 に画像がアップロードされたことをトリガーにして Lambda が実行される
- Lambda でサムネイルを作って S3 に保存

S3 の URL を保存するには、Lambda 内でデータベースに登録することもできるし、ファイルをアップロードする代わりに URL を渡すエンドポイントを作ってもいいでしょう。

これだけでも、登場人物は **Cognito**, **S3**, **Lambda** と、裏で重要な働きをしている **IAM** と、どう実装されてるかさっぱりわからない **トリガー** がいます。小さな機能が有機的に繋がってシステムを作り上げる感覚です。

大きな違いは、サーバとファイルのやり取りをしないことです。サーバがファイルを扱わないので、余計な通信も発生しないし、サーバの I/O 待ち時間も減らせます[*3]。

難点としては、やはり AWS の広い知識が求められることです。実現したい機能のためにはどのような AWS のサービスが利用できるのか、従来の方法で使ってきた範囲の AWS の知識でとは**まるでかすりもしない**ということも多々あります。今まで存在すら知らなかったサービスを、ひとつひとつ調べることになるかもしれません。特に、従来の方法では結構雑でも問題なかった **権限設計** がとても重要になるので、 IAM の高い理解が必要になります。

[*2] 例: paperclip など

[*3] 従来はクライアント ↔ サーバ ↔ S3 と 2 回巨大ファイルを通信しないといけない

第 2 章

初めての Lambda プログラミング

まずはシンプルに、AWS のコンソール[*1] から初めての Lambda プログラミングをしてみましょう。先程のサムネイルを作るプログラムを作りたいのですが、まずは標準ライブラリだけでできる内容を例にしたいと思います。**S3 にファイルが追加されたら指定された URL に POST するプログラム** を作ってみましょう。つまり Webhook ですね。

Create function ボタンを押し、スクラッチから作る (Author from scratch) を選びます。適当な関数名を入力し、 **ランタイムに Ruby2.5 を選択** します。Lambda を実行するロール (Execute Role) は、まだ存在しないので自動作成を選びましょう。Lambda 関数を作成すると、関数の編集ページに遷移します。

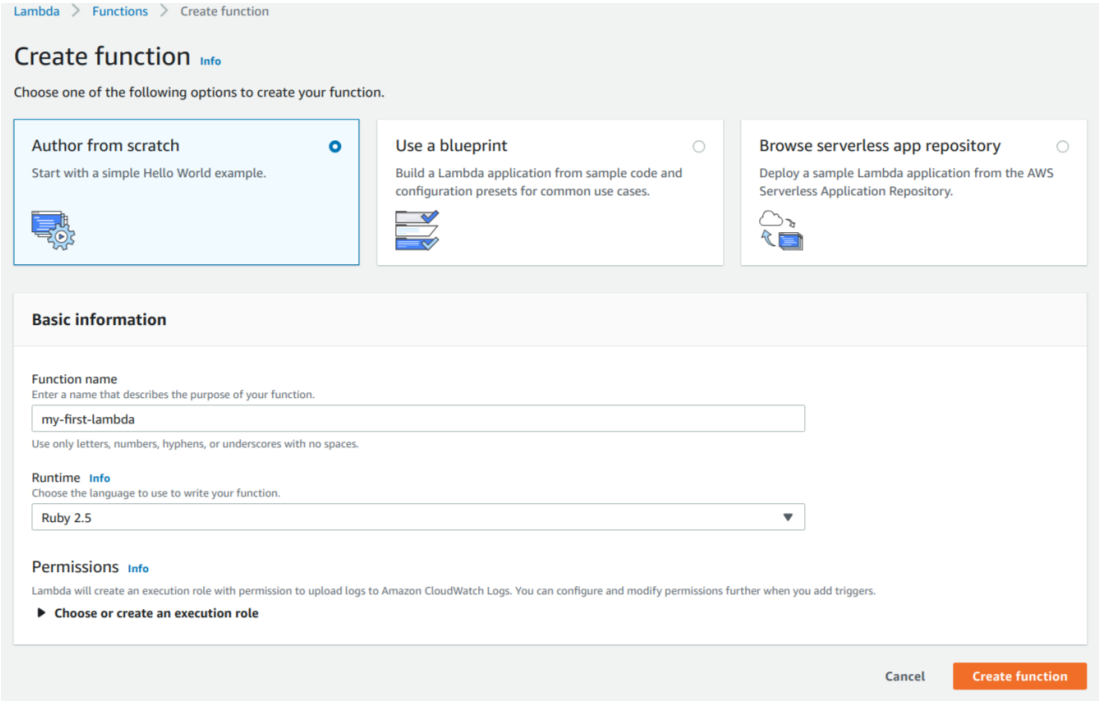

図 2.1: Lambda 作成画面

[*1] https://console.aws.amazon.com/lambda

編集ページの下の方には、ロール設定やネットワーク設定などの細々した設定メニューがあります。重要なのは一番上にある **Designer** とその次の **Function code** です。Designer では、Lambda を実行する **トリガー** を GUI で設定することができます。Function Code はもちろんコードを書く部分ですね。そしてもう一つ大事な機能、一番上のグローバルメニューから **関数のテスト** が実行できます。

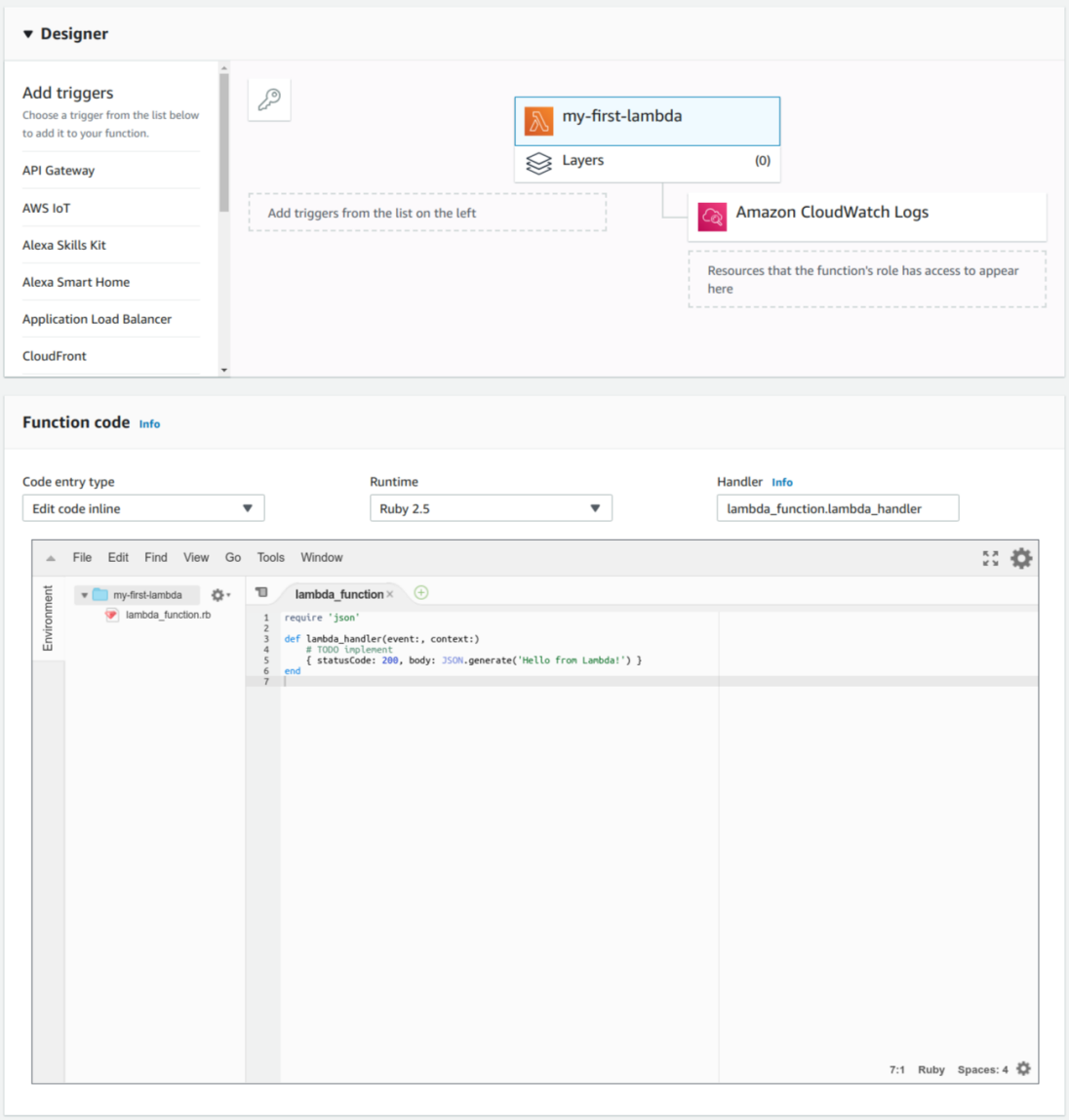

図 2.2: Lambda のエディタ

2.1 トリガーを指定してみる

今回トリガーにするのは **S3 にファイルが追加された時** なので、 左のリストから S3 を探してクリックします。Designer 画面に S3 が追加され、 **Configuration requried** という警告が出ています。すぐ下にトリガーの編集画面があるので設定していきます。

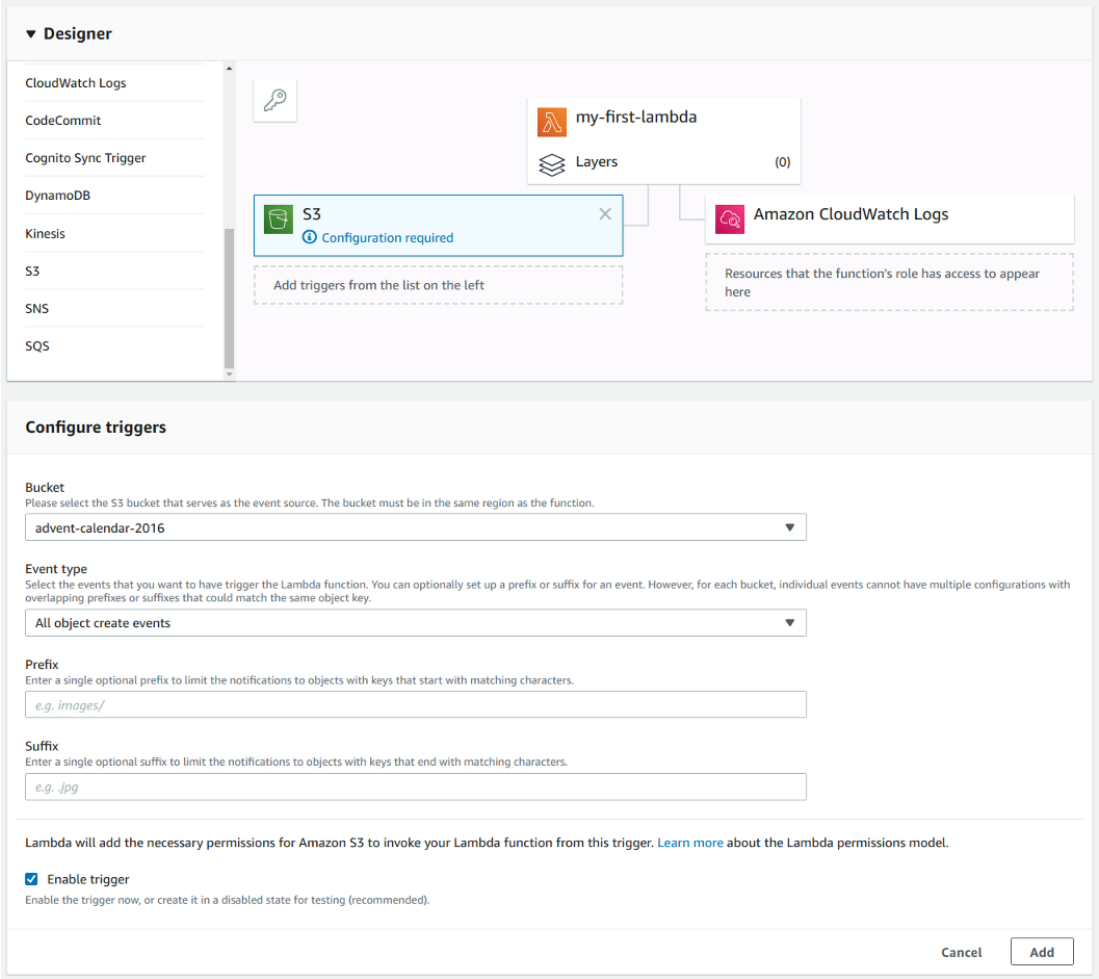

図 2.3: S3 トリガーの設定画面

対象のバケット名、イベントタイプは `All object create events` を選びます。`Prefix` や `Suffix` も選べるので、例えば Prefix に `/upload/source` を指定することで、 `/upload/source` に追加された画像のサムネイルを `/upload/thumbnail` に出力すると言ったことが可能になります。設定が終わったら `Add` ボタンを押し、いったんグローバルメニューから `Save` しましょう。

2.2 テストを実行してみる

次にざっとコードを見てみます。ウィザードで作られた状態では、 `lambda_function.rb` と言うファイルに `lambda_handler` というメソッドが定義されています。Handler の欄には `lambda_function.lambda_handler` と指定されているので、**ファイル名. メソッド名** が呼ばれるのだろうと推測できます。

メソッド定義を見てみると、 `event` と `context` の引数を取り、 `statusCode` と `body` のハッシュを返していることがわかります。さて、これをいじっていくのですが、まずこの 2 つの引数がどんなものなのかわからないと開発しようがありません。そこでまずテストを実行してみることにします。

Test のボタンを押すと、まずテストイベントを作ります。プリセットがたくさんあるので、該当しそうなものを探します。S3 に関しては **Amazon S3 Put** と **Amazon S3 Delete** が用意されています。おそらく Put を選べば良さそうなので、そのまま任意の名前をつけてイベントを作ります。なお、Lambda 開発しているとよくあるのですが、**本当にそのイベントであっているのか** 実際に本物のイベントが飛んでくるまでわかりません。S3 は運良くプリセットがあったのですが、サービスによっては **プリセットが用意されてない** 場合もあります。**ぶっちゃけ Lambda はここが一番難しい** と思います。

あとは普通の Ruby 開発のように、 `p event:event, context: context` な感じにデバッグ出力を入れて実行してみましょう。Execution Result にログ出力されているのが見つかると思います。

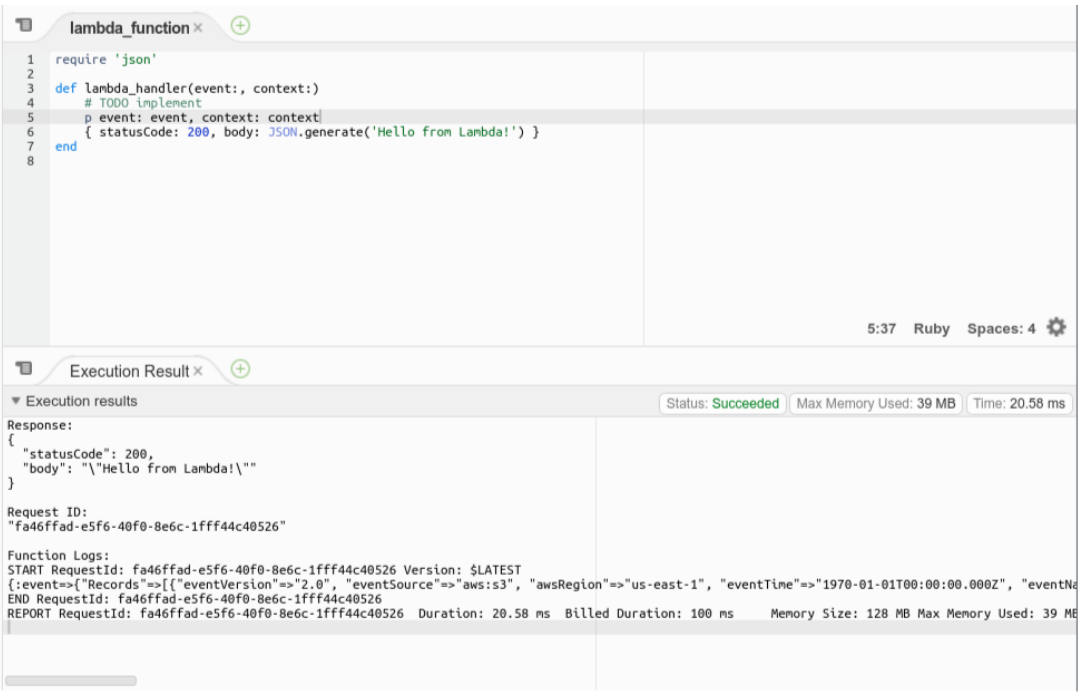

図 2.4: ログ出力

2.3　本当にそれでいいの・・・?

イベント、本当にわからない。やっぱりもうこの段階から **本物のイベント** を動かしたほうが良さそうな気がします。実際に選んだ S3 にファイルを置いてみましょう。どこにログが出力されるかも、Designer をみればわかります。**Amazon CloudWatch Logs** ですね。

デプロイごと (Save を押すごと) にログがわけられているので、最新のログを探します。その中から本物のログ出力を見つけます。実物のバケット名が入っていたらおそらくそれです。比べてみると、**Amazon S3 Put** と同じ形式なのでそのままで大丈夫なことがわかりました。何なら本物のイベントでテストの JSON を書き換えても良いかもしれません。

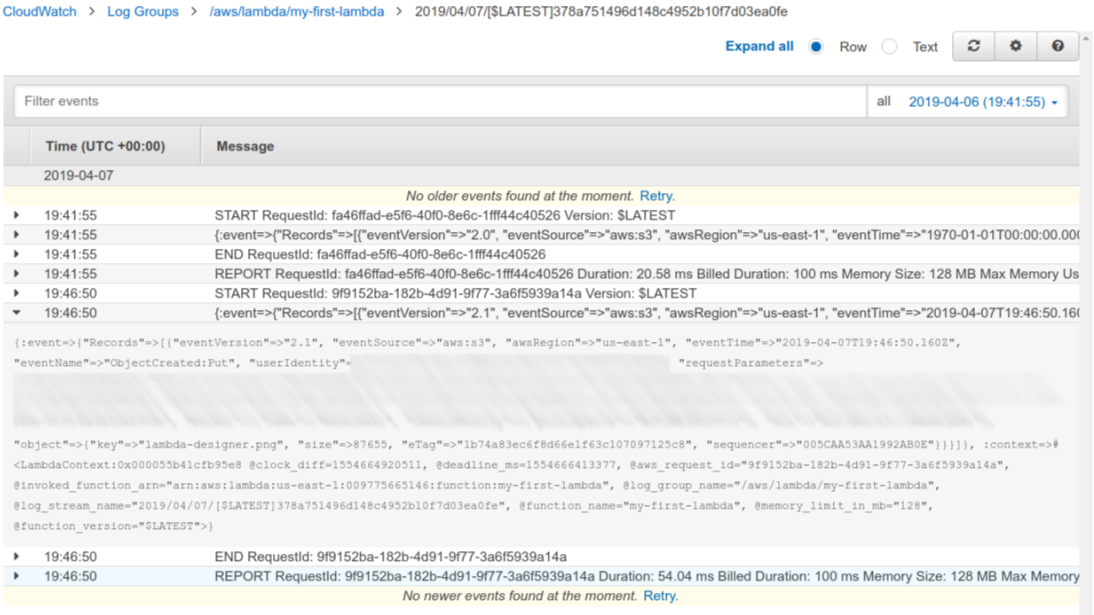

図 2.5: CloudWatch Logs に出力されたログ

2.4　コードをいじる

`event` と `context` が分かったので後はみんな大好き普通の Ruby です。

```
require 'json'
require 'uri'
require 'net/http'

def lambda_funciton(event:, context:)
  event['Records'].each do |record|
    key = record['s3']['object']['key']
    uri = URI(ENV['WEBHOOK_URL'])
```

```
    payload = JSON.generate(text: "s3: #{key} created.")
    Net::HTTP.post_form(uri, payload: payload)
  end
  {statusCode: 200, body: ''}
end
```

テストを実行したり実際にファイルをアップロードしたりしてみましょう。環境変数の設定は Function code の下にあります。

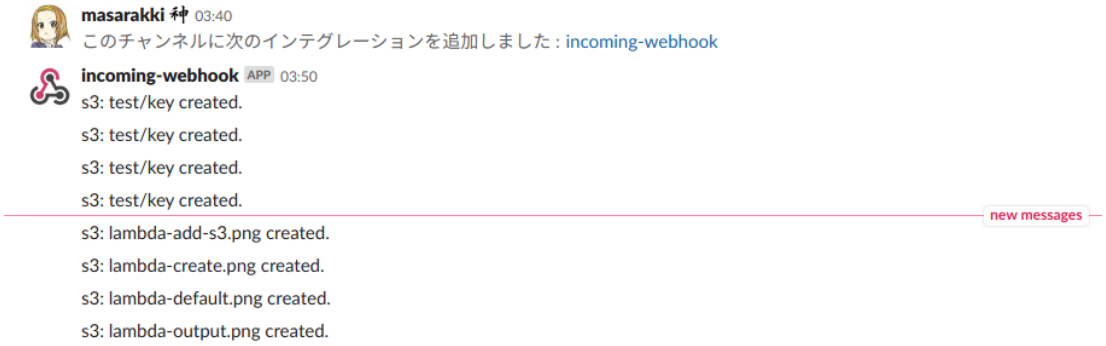

図 2.6: Webhook を受信

上手く動きました!! 完成したコードは samples/first-lambda[*2]にあります。

[*2] https://github.com/np-complete/TechBookFes06/tree/master/samples/my-first-lambda

第 3 章

AWS SAM を使ってみる

AWS のコンソールで簡単な Lamda を動かすことができました。コンソール上でも複数ファイルを作ってライブラリのように読み込むこともできるし、同様に gem も置けます[*1]。何だってできそうな感じがしますが、いやどう考えてもコンソールはいずれ破綻しますよね。我々はプログラマなので、**Git 管理が可能** なテキストベースのファイルと CLI ツールがないと死んでしまいます。そこで、**AWS SAM**(Serverless Application Model) というものを使ってみます。

3.1 AWS SAM とは

AWS SAM は Lambda でサーバレスアプリケーションを作るためのフレームワークです。似たようなものに、Javascript 界隈で先行した severless[*2] や、Ruby だと最近出た Ruby on Jets[*3] がありますが、AWS SAM はその名の通り AWS 公式ツールなのが大きな強みです。

AWS SAM は、実際には **Cloud Formation** をベースにした Lambda の構成管理とデプロイのツールです。Cloud Formation なので **人類にはかなり難しい** んですが、Lambda くらいの範囲だとそこそこシンプルなのでなんとかなりそうです。むしろ破綻しないくらいのシンプルなプログラムを目指しましょう。

3.2 初めての SAM プログラム

先程の Lambda を SAM に移植していきましょう。まず SAM の CLI ツールをインストールします。

```
pip install aws-sam-cli
```

インストールが完了したらおもむろにプロジェクトを作ります。

[*1] `Gemfile` は読まないので `bundle exec` ではないらしい
[*2] https://serverless.com/
[*3] http://rubyonjets.com/

```
sam init -r ruby2.5 -n my-first-sam
```

-r でランタイムを指定できます。Ruby のランタイムを指定して実行するときちんと Ruby のファイルが作られます。my - first-sam ディレクトリが作られ、中には template.yaml と hello_word ディレクトリが作られます。template.yaml が Cloud Formation の構成ファイルで、hello_world の中身が実際の Lamda 関数になっています。hello_world ディレクトリの中には Gemfile も存在します。ススっと template をいじっていきます。

```
AWSTemplateFormatVersion: '2010-09-09'
 Transform: AWS::Serverless-2016-10-31
 Resources:
   HelloWorldFunction:
     Type: AWS::Serverless::Function
     Properties:
       CodeUri: hello_world/
       Handler: app.lambda_handler
       Runtime: ruby2.5
       Environment:
           Variables:
               WEBHOOK_URL: https://example.com/webhook
```

hello_world/app.rb の中身も前回のもので置き換えます。おもむろにデプロイします。

```
sam package --output-template-file packaged.yaml \
            --s3-bucket lambda-packages
sam deploy --template-file packaged.yaml \
           --stack-name my-first-sam \
           --capabilities CAPABILITY_IAM
```

package コマンドでは自動的に必要なファイルがパッケージ化され、 --s3-bucket で指定したバケットにアップロードされます。deploy コマンドで実際に Cloud Formation のスタックにデプロイされ、変更が反映されます。現時点では、まだトリガーとなる S3PUT の設定がありませんが、他の部分は前回のと同じ状態になっているはずです。

3.3 トリガーを設定する

まず Resources に対象となる S3 バケットを作る設定をします。AWS::S3::Bucket のドキュメント[*4] を見ながら埋めていきましょう。実は、必要なプロパティは一つもないので Resources の下に

[*4] https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-s3-bucket.html

```
Bucket:
    Type: AWS::S3::Bucket
```

を追加するだけで完成です。 これで Bucket と言う名前で参照できるようになります。この時点でデプロイしてみて S3 バケットが作成されるのを確かめましょう。名前を指定していないので、my-first-sam-bucket-(ランダム文字列) というバケットができました。

次にイベントの設定です。HelloWorldFunciton のプロパティにイベントを追加します。S3 のイベントドキュメント[*5] を見て設定しましょう。

```
Events:
    Uploaded:
        Type: S3
        Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

デプロイ実行してみてもトリガーのところには何も表示されません。しかし、実際に対象バケットにファイルを追加してみると、トリガーが発火し Lamda が実行されていることがわかります。少し困ったことに、バケット名を直接指定することはできないようで、必ず同じテンプレート内で作ったものを参照しなければならないようです。

最終的にはこのようなテンプレート[*6] になりました。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: my-first-sam

Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: hello_world/
      Handler: app.lambda_handler
      Runtime: ruby2.5
      Environment:
        Variables:
          WEBHOOK_URL: https://example.com/webhook
      Events:
        Uploaded:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
```

[*5] https://github.com/awslabs/serverless-application-model/blob/master/versions/2016-10-31.md#s3
[*6] https://github.com/np-complete/TechBookFes06/blob/master/samples/my-first-sam

```
          Events: s3:ObjectCreated:*
Bucket:
  Type: AWS::S3::Bucket
```

次は SAM で gem を使ったり複雑なコードにしていきます。

第 4 章

SAM で Gem を使ってみる

Ruby を書くなら当然便利な Gem を使いたくなりますね。SAM の `hello_world` ディレクトリには最初から `Gemfile` が入っているので、やはりこれを使えということなんでしょう。とりあえず適当な Gem を使ったコード[*1]を書いてみましょう。特に意味のあるコードにする必要はないので、ただ `require` して簡単なコードを実行するだけのコードでいいと思います。

```
# hello_world/Gemfile

source 'https://rubygems.org'
gem 'faraday'

# hello_world/app.rb

require 'json'
require 'faraday'

def lambda_handler(event:, context:)
  Faraday.get('https://google.com')
  { statusCode: 200, body: '' }
end
```

この段階でおもむろにデプロイしてテストを実行すると、 `cannot load such file -- faraday` とエラーが出ます。Gem がインストールされていないようですね。`Gemfile` を用意していても自動的に `bundle install` をしてくれたりは無いようです。

とりあえず SAM のパッケージの中に全部入れれば良さそうなので、いつもどおりの `bundle install --path vendor/bundle` をしてみます。デプロイしてコンソールを確認すると、ファイル一覧の中に `vendor` ディレクトリがあり、Gem が配置されていることがわかります。テストを実行してみると、何もエラーなく終了しました。

[*1] https://github.com/np-complete/TechBookFes06/blob/master/samples/sam-with-gem

4.1 Gem を読み込むのはそんなに簡単ではないという話

Gem の場所を指定する超重要ファイルである `.bundle` が無いように見えますが、隠しファイルを表示するとちゃんと出てきます。手元から `.bundle` を削除し、デプロイしてみましょう。実行してもエラーは起こりません。更にもっと超重要ファイルである `Gemfile` と `Gemfile.lock` も消してみましょう。なんと、**Gemfile がなくても成功します**。これはかなり不思議な動作です。とりあえず `bundler` を使っていないのは確実なようです。なぜ `vendor/bundle` 以下を Gem として読めているのでしょか、探ってみましょう。

`require` のコードから辿っていくと、最終的に Gem のリストは `Gem::Specification.stubs` に格納されるようです。`Gem::Specification.stubs.map(&:name)` を実行してみると、明示的にはインストールしていない `aws-***` と並び、確かに `faraday` は存在します。`gems_dir` を見てみると `/var/tasks/vendor/bundle/ruby/2.5.0/gems` になっています。`Gem::Specification.dirs` にもやはり該当パスが存在し、`Gem.paths` を辿って `Gem::PathSupport` というところで定義されていることがわかりました。

コードを読んでみると `ENV['GEM_PATH']` が重要らしいので出力してみると、まさにドンピシャ、`/var/task/vendor/bundle/ruby/2.5.0:/opt/ruby/gems/2.5.0` となっていました。どうやら Lambda のデフォルトの環境変数でこう設定されているようです。

最後に、おもむろに `vendor/bundle` を削除します。それでも `ENV['GEM_PATH']` には `vendor/bundle` が存在し続けています。つまり、`ENV['GEM_PATH']` に `vendor/bundle` が含まれているのは完全に固定の値であり、**いつもどおりやったら Gem が使えたのはただの偶然** だと言えます。その証拠に、 `bundle install --path gems` のように `vendor/bundle` 以外を指定した場合、**Gem の読み込みは失敗します**。このあたり暗黙のルールが完全に共通認識になっている感じ、**とても Ruby らしい** と思いました。というわけで、Gem は `vendor/bundle` に置くか、`/opt/ruby/gems/2.5.0` に置けば良いことがわかりました。

4.2 C 拡張が入った Gem を使う

Ruby には C 言語インターフェイスが用意されており、C 言語拡張が入った Gem を作ることができます。そのような Gem を `gem install` すると、自動でコンパイルが走り、所定の場所に `.so` ファイルが作られます。今回は C 拡張を含む Gem を、

- 単純に高速化のために C 拡張を使う (例: `redcarpet`)
- 外部の C のライブラリを Ruby から使うために C 拡張を使う (例: `mysql2`)

の 2 種類に分割して考えます。ようは、外部ライブラリを使うか使わないかで難しさが変わるためです。

まず、先程見たように Lambda 内で `bundle install` はしてくれないので、**.so ファイルもパッケージに含める** 必要があります。となると、アーキテクチャの問題が出てきますね。まず一番雑に手元で `bundle install` してみましょう。`vendor/bundle/ruby/2.5.0/gems/redcarpet-3.4.0/lib/redcarpet.so` が存在することを確

認しておきましょう。

コードはこんな感じです。

```
require 'json'
require 'redcarpet'

def lambda_handler(event:, context:)
  md = Redcarpet::Markdown.new(Redcarpet::Render::HTML)
  { statusCode: 200, body: JSON.generate(text: md.render('hello, **world**')) }
end
```

これをそのままデプロイすると、`require 'redcarpet'` が失敗しエラーになります。非常に重要なヒントが書いてあり、

```
Ignoring redcarpet-3.4.0 because its extensions are not built.
```

つまり、ビルドに失敗しているという判定がなされているようなのです。もちろん.so ファイルは存在するので、想定していたとおりアーキテクチャの違いなどで読み込めていないと予想できます。

`cat /etc/*-release` を実行すると、ディストリビューションは **Amazon Linux AMI** であることがわかります。Amazon Linux は RedHat 系なので、Ubuntu でビルドしたバイナリが使えないのはそういうことなのでしょう。なんとかして AmazonLinux でビルドをしないといけないのですが、なんと Amazon Linux は Docker イメージが提供されています。このイメージに Ruby をインストールし、ビルド用イメージを作ってみましょう。できあがったイメージが masarakki/aws-lambda-ruby[*2] です。

このイメージ内の `bundle` を使って、ホストマシンに gem をインストールします。

```
$ rm -r vendor/bundle
$ docker run -it -v `pwd`:`pwd` -u `id -u`:`id -g` -w `pwd` \
             masarakki/aws-lambda-ruby \
             bundle install --path vendor/bundle
```

これで Amazon Linux ベースのマシンでビルドした `redcarpet` が入手できたはずです。`bundle exec irb` を実行してみると、

```
Could not find redcarpet-3.4.0 in any of the sources
```

[*2] https://github.com/masarakki/aws-lambda-ruby

と言われ、先ほどと立場が逆転していることがわかります。これは期待できそうです。早速デプロイしてテストを実行してみたところ、見事に成功しました。

なお、双方の.so ファイルを `file` コマンドで見たところ、どちらも **64bit ELF** なので、アーキテクチャの問題ではなさそうです。依存ライブラリの問題ではないかと疑い `ldd` してみたら、ライブラリの数まで違っていたので、おそらくこれが原因なのでしょう。ライブラリの配置場所やファイル名は基本的に一致しているようでした。関数のアドレスが違うのでは? と言う指摘をされましたが、正直 C の素養が全然ないのでよくわかりません。こういうところで問題が出るということを知れたことは新鮮です。

ということは、**クロスコンパイル** では問題は解決しないと予想できます。また、Lambda 内部の Amazon Linux がバージョンアップなどで変化した場合も動かなくなる可能性があります。これはかなり茨の道なのではないかと感じました。Lambda が動いている実体コンピュータのバージョン指定的なオプションが欲しいと思いました[*3]。

4.3 外部ライブラリを使う Gem を使う

次に、外部ライブラリを使う Gem を試してみましょう。ただでさえ難しかった.so ファイルが、今度はシステムの別の場所にもあるので更に難しくなります。とはいえ、こちらも難易度に 2 段階あり、まずは簡単な **既にシステムにインストールされている** ライブラリを使った Gem に挑戦します。Amazon Linux には最初から様々なライブラリがインストールされているので、`/usr/lib64` を眺めながらインストールできそうな Gem を探します・・・・・・・`nokogiri` しか見つかんねえ。

どうせなら前回のコードを活かして `nokogiri` を使ってみましょう。

```
require 'json'
require 'redcarpet'
require 'nokogiri'

def lambda_handler(event:, context:)
  md = Redcarpet::Markdown.new(Redcarpet::Render::HTML)
  html = Nokogiri::HTML(md.render('hello, **world**'))
  { statusCode: 200, body: JSON.generate(text: html.css('strong').text) }
end
```

デプロイしてみるとエラーなく終了し、

```
{"statusCode": 200, "body": "{\"text\":\"world\"}"}
```

と出力されます。上手く動いているようです。しかし、このあたりからバイナリを含むせいでパッケージサイズが大きくなり、コンソール上でコードが確認できなくなるのがつらいところです。

[*3] ちがうそうじゃない 頼むから `bundle install` させてくれ

4.4 インストールされてない外部ライブラリを使う

さていよいよ、一番の難関になりそうな **インストールされてないライブラリ** を使う Gem に挑戦しましょう。とりあえず postgres のコネクタである pg をインストールしてみます。もちろん libpq は入ってないのでそのままではインストールできません。

そこで、まずは libpq をインストールすることを考えます。

```
docker run it -v ‘pwd‘:‘pwd‘ -w ‘pwd‘ masarakki/aws-lambda-ruby /bin/bash
```

で docker イメージに入り、

```
yum install postresql-devel
```

でライブラリをインストールします。あとは、

```
bundle install --path vendor/bundle
```

で Gem をインストールします。この方法は少し問題があり、Gem が root でインストールされてしまいますが、今のところどうしようもありません。

ホストコンピュータに戻り ldd で依存ライブラリを見てみます。libpq.so.5 => /usr/lib/x86_64-linux-gnu/libpq.so.5 (0x00007fdaa32d1000) の一行が入っているのが見つかります。おそらくこのまま持っていっても Lambda の環境には libpq.so.5 が無いので動かないでしょう。この時点でデプロイして試したところ、やはり pg が見つからないというエラーになります。とはいえ、前回のようにビルドできてないからインストールしろというメッセージではなく、普通に無いと言われています。ちょっと予想外の反応です。

さーこれどうしましょうか・・・Lambda は /var/task に単純にファイルが展開されるだけなので、 /usr/lib の下に libpq.so を置くことは難しそうです。

第 5 章

Layer とカスタムランタイム

ライブラリを設置できない問題に対し、Lambda の **Layer** という機能が使える可能性があるので調べます。ドキュメント[*1] によると、Layer は **ライブラリ、カスタムランタイム、その他の依存をまとめた ZIP アーカイブ** という定義のようです。もしかしたらこれで `libpq.so` が配置できるかもしれませんが、ドキュメントによるとどうやら主目的は毎回のデプロイサイズを小さく保つことのようです。

ドキュメントを探っていくと、それぞれのランタイム向けにどのようにファイルを配置するか書いてあります。例えば Ruby は、 `ruby/gems/2.5.0/` の中に Gem の構造のファイル群を入れれば良いようです。また、どのランタイムに対しても `lib/` がライブラリに対応すると書いてあります。早速試してみましょう。

5.1 シンプルな Ruby のライブラリを Layer 化する

まず単純に、 `ruby/lib` に単純な Ruby のコードを置き、`app.rb` から読み込めるかどうか実験してみましょう。

```
mkdir -p layers/simple-layer/ruby/lib
cd layers/simple-layer
# ruby/lib/hello.rb に Hello クラスを作る
zip -r simple-layer.zip ruby
```

zip コマンドクソ難しいですね。

```
unzip -l simple-layer.zip
Archive:  simple-layer.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2019-04-13 13:26   ruby/
        0  2019-04-13 13:27   ruby/lib/
```

[*1] https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html

```
      101  2019-04-13 13:27   ruby/lib/hello.rb
---------                     -------
      101                     3 files
```

確認すると正しいディレクトリ構造になっていそうです。この simple-layer.zip をとりあえず S3 にアップロードします。

```
aws s3 cp simple-layer.zip s3://masarakki-sam-deploy/simple-layer-v1.zip
```

Cloud Formation で AWS::Serverless::LayerVersion のリソースを作ってみましょう。なお、Layer のバージョンを上げるには ContentUri の s3 ファイル名を変更するしかありませんでした。

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: hello_world/
      Handler: app.lambda_handler
      Runtime: ruby2.5
      Layers:
        - !Ref SimpleLayer
  SimpleLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      ContentUri: s3://masarakki-sam-deploy/simple-layer-v1.zip
      CompatibleRuntimes:
        - ruby2.5
```

app.rb はシンプルにこんな感じにしました。

```
require 'json'
require 'hello'

def lambda_handler(event:, context:)
  {
    statusCode: 200,
    body: {
      message: Hello.new('world').say
    }.to_json
  }
end
```

実行すると正しく動作しました。

5.2 共有ライブラリを Layer 化する

さて、次は pg の Gem に必要なライブラリを Layer にしてみましょう。ひとまず、layers/pg-layer/lib を作り、その中で Docker イメージにログインします。

```
mkdir -p layers/pg-layer/lib
cd layers/pg-layer/lib
docker run -it -v ‘pwd‘:‘pwd‘ -w ‘pwd‘ amazon-linux /bin/bash
```

Docker イメージの中に入ったら postgres のライブラリをインストールし、インストールされたファイルをコピーします。

```
yum install postgresql-devel
cp /usr/lib64/libpq* .
```

これでホストマシンにライブラリがコピーされています。先ほどと同じように ZIP ファイルを作り S3 にアップロードし、リソースを作ります。

```
AWSTemplateFormatVersion: ’2010-09-09’
Transform: AWS::Serverless-2016-10-31
Description: sam-with-layers

Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: hello_world/
      Handler: app.lambda_handler
      Runtime: ruby2.5
      Layers:
        - !Ref SimpleLayer
        - !Ref PgLayer
  SimpleLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      ContentUri: s3://masarakki-sam-deploy/simple-layer-v2.zip
      CompatibleRuntimes:
        - ruby2.5
  PgLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      ContentUri: s3://masarakki-sam-deploy/pg-layer-v1.zip
```

`app.rb` の方は適当なデータベースに繋ぎに行くコードにしてみましょう。アプリケーションの Gem は前回と同様の方法でインストールします。デプロイし実行してみると、

```
could not connect to server: No such file or directory
```

というエラーが出ます。これは、`pg` の Gem が無事にロードでき、さらに **実際に connect を試みて** 失敗している、つまり確実に `libpq.so` に到達していると言えるでしょう。実験は大成功です!! 最終的なコードは samples/sam-with-layer[*2]にあります。

AWS Lambda のマシンが持っていないライブラリでも、Layer を使いライブラリを渡すことによって、Gem が使えることがわかりました。ひとつひとつの Lambda 関数を小さく保つために、共有ライブラリだけではなく、C 拡張をビルドするようなファイルサイズが大きな Gem も、Layer 側に寄せていく方が良いという思想のようです。しかし、Layer はできたての機能らしく、作業ログを見てわかるようにまだ **泥臭い** 作業をしなくてはならず、今回はアプリケーション側に持たせることにしました。おそらく時間が経てばこのあたりも改善されていくことでしょう。個人的には

```
PgLayer:
  Type: AWS::Serverless::LayerVersion
  Properties:
    Yum:
      - postgresql-devel
    Gem:
      - pg
```

みたいに書いたら上手いこと配置してくれるような DSL が欲しいですね。

・・・・あれ? カスタムランタイムの話無くない?

[*2] https://github.com/np-complete/TechBookFes06/tree/master/samples/sam-with-layers

あとがき

体調不良で 2 日ロスし、絶対ムリだと思ったけどなんとか間に合いました。

軽く死にかけました。

原稿で寝不足になっていたところ、徹夜作業中にコーヒーを飲んだら心臓が爆発しそうになりました。激しい動悸、顔面蒼白、目の前ぐらぐら、胸焼けに止まらない脂汗、脳が **こんな辛いなら生きるのを諦めたほうが良い** という判断を出している感覚がありました。

ついにぎっくり腰も経験し、30 代も半ばになるとちょっとした体調不良で **体は生きるのをやめようとします**。マジで健康が何より重要。ちゃんと寝てください。バランスの良い食事をしっかり摂って。水分も意識的に過剰と思えるくらい摂ったほうが良いです。適度に運動もしましょう。あとしっかり寝て。

マジで健康は重要です。

そういえば、Gitbook が無料で全然使えなくなったので別の方法で電子書籍版の配信をしないといけません。今回は一旦 S3 から PDF をダウンロードできるようにしましたが、やはり HTML で読めてほしいので何かシステム作らないといけないのかなぁ〜と考えてます。

Ruby on AWS Lambda

2019 年 4 月 14 日　初版第 1 刷　発行
著　者　まさらっき
発行所　NP-complete
印刷所　キンコーズ