AWS - Lambda and selenium

What ?

I have recently started a slow and awkward dance with Amazon and their serverless solutions. And no, I did not forget you OpenFaaS, daddy still loves you.

Long story short, I decided to automate one of my tasks I often do in work and to learn Lambda at the same time. This ended up quite a journey, which was successful in the end but almost died in its infancy since the first step I needed to automate was to Log In to one specific page which have very "cleverly" made log in process and can't be done via requests library. But I knew that it can be done via Selenium and Chromedriver.

Issue

Lambda functions in AWS are great and if Synthetic Canaries could return actual custom value instead only failed / not failed I would not have to bother with all this., The issue is that you are limited with space of how much you can deploy into single lambda function and that's 250 MB.

Remember, you need to squeeze:

- ***Your script*** and any additional library you need, just few Kb
- *Selenium* few Kb
- *Chromedriver*: cca 20MB
- *Browser itself* "chrome-linux: whooping 405 MB !

You see the issue now ? 250 MB vs 425 MB+

You can try to fit it in with layers, and I was able to get Selenium, Chromedriver and Chrome into Lambda with layers... but! In the background, it still runs in a small container that simply does not have dependencies for Chrome browser. You will end up with:

> **Bug**
>
> chromedriver unexpectedly exited. Status code was: 127

A solution

Well since there is no way to fit it to pure Lambda there is another Lambda option and that is to use your own container. Here the limits are a bit more relaxed. Your container can be max 10 GB in size.

So I assume you have some basic knowledge about the Lambda, AWS cli, or SAM (I sure did not know I can use my own container as lambda function...)

Creating basic function

Let's look at basic file and folder structure of simple Lambda function that is able to invoke Chromedriver and Chrome. I'm going to use SAM to generate scaffolding for this function.

```
vladoportos@Odin:~/amazon_aws/test$ sam init
Which template source would you like to use?
        1 - AWS Quick Start Templates
        2 - Custom Template Location
Choice: 1

Cloning from https://github.com/aws/aws-sam-cli-app-templates

Choose an AWS Quick Start application template
        1 - Hello World Example
        2 - Multi-step workflow
        3 - Serverless API
        4 - Scheduled task
        5 - Standalone function
        6 - Data processing
        7 - Infrastructure event management
        8 - Machine Learning
Template: 1

 Use the most popular runtime and package type? (Nodejs and zip) [y/N]: N

Which runtime would you like to use?
        1 - dotnet5.0
        2 - dotnetcore3.1
        3 - go1.x
        4 - java11
        5 - java8.al2
        6 - java8
        7 - nodejs14.x
        8 - nodejs12.x
        9 - python3.9
        10 - python3.8
        11 - python3.7
        12 - python3.6
        13 - ruby2.7
Runtime: 9

What package type would you like to use?
        1 - Zip
        2 - Image
Package type: 2

Based on your selections, the only dependency manager available is pip.
We will proceed copying the template using pip.

Project name [sam-app]: sample-app

    -----------------------
    Generating application:
    -----------------------
    Name: sample-app
    Base Image: amazon/python3.9-base
    Architectures: x86_64
    Dependency Manager: pip
    Output Directory: .
    Next steps can be found in the README file at ./sample-app/README.md


    Commands you can use next
    =========================
    [*] Create pipeline: cd sample-app && sam pipeline init --bootstrap
    [*] Test Function in the Cloud: sam sync --stack-name {stack-name} --watch
```

This will generate basic folder structure and files. Important part is to chose Image for package type.

```
vladoportos@Odin:~/amazon_aws/test/sample-app$ ls
README.md
__init__.py
events
hello_world
template.yaml
tests

#cd to hello_world
vladoportos@Odin:~/amazon_aws/test/sample-app/hello_world$ ls
Dockerfile
__init__.py
app.py
requirements.txt
```

Above you can see the folder structure generated for us.

Most important files are:

- ***template.yaml*** - Telling SAM how to deploy our function.
- ***hello_world/Dockerfile*** - Docker file, we are going to change this.
- ***hello_world/app.py*** - Our function, I will show you what to add there.
- ***hello_world/requirements.txt*** - Classic requirement file for python app, add your libraries there, so they will be automatically added to your app.

template.yaml

Change it to look like this:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: >
  python3.9

  Sample SAM Template for sample-app

Globals:
  Function:
    Timeout: 60
```

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      PackageType: Image
      MemorySize:  512
      Timeout: 300
      Architectures:
        - x86_64
      Environment:
        Variables:
          URL: 'https://google.com'
    Metadata:
      Dockerfile: Dockerfile
      DockerContext: ./hello_world
      DockerTag: python3.9-v1
```

I have removed API and Outputs from it, we don't need that now.

> **Note**
>
> - If you want to rename your function, change the "HelloWorldFunction" here as well. Also, the name of folder where you function live can be changed here under "DockerContext: ./hello_world" just don't forget to rename the folder as well 😃
> - I have added an environment variable with URL that I will refer to in the code later.
> - Also I have increased RAM for that function to 512 MB

requirements.txt

I have just Faker in it to generate random user-agent for the Chromedriver. Its nifty library to generate fake data that looks convincing 😃 Look into its documentation here Faker Documentation

```
Faker==6.4.1
```

> **Note**
>
> Notice I did not put selenium in there... spooky, right ?

Dockerfile

Our pièce de résistance, let's look at it:

```
#based on https://github.com/umihico/docker-selenium-lambda
FROM public.ecr.aws/lambda/python:3.8 as build

RUN yum install -y unzip && \
    curl -Lo "/tmp/chromedriver.zip" "https://chromedriver.storage.googleapis.com/98.0.4758.48/chromedriver_linux64.zip" && \
    curl -Lo "/tmp/chrome-linux.zip" "https://www.googleapis.com/download/storage/v1/b/chromium-browser-snapshots/o/Linux_x64%2F950363%2Fchrome-linux.zip?alt=media" &&
\
    unzip /tmp/chromedriver.zip -d /opt/ && \
    unzip /tmp/chrome-linux.zip -d /opt/

FROM public.ecr.aws/lambda/python:3.8
RUN yum install atk cups-libs gtk3 libXcomposite alsa-lib \
    libXcursor libXdamage libXext libXi libXrandr libXScrnSaver \
    libXtst pango at-spi2-atk libXt xorg-x11-server-Xvfb \
    xorg-x11-xauth dbus-glib dbus-glib-devel -y

RUN pip install selenium

COPY --from=build /opt/chrome-linux /opt/chrome
COPY --from=build /opt/chromedriver /opt/

COPY app.py requirements.txt ./

RUN python3.8 -m pip install -r requirements.txt -t .

# Command can be overwritten by providing a different command in the template directly.
CMD ["app.lambda_handler"]
```

> **Note**
>
> I used Python3.8 image despite choosing function for Python3.9, for simple unverified reason that I have read some comments on internet that Python3.9 had some issue with selenium and Chromedriver. This is completely unverified, so try 3.9 first and if there are issues switch to 3.8.

I used the official python3.8 image from Amazon, and extended it with all we need. This was based on this repo: Github Repo, credit where credit is due.

You can see that it download the Chromium and Chromedriver, unzip and put them to /opt install selenium etc..

app.py

In our function, we need to keep some special setting for chrome to have it run ok.

```
from tempfile import mkdtemp
import logging
import sys
import os
from faker import Faker
import random
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import NoSuchElementException, WebDriverException, TimeoutException, NoSuchWindowException


language_list = ['en']

#This part make logging work locally when testing and in lambda cloud watch
if logging.getLogger().hasHandlers():
    logging.getLogger().setLevel(logging.INFO)
else:
    logging.basicConfig(level=logging.INFO)


#Function that setup the browser parameters and return browser object.
def open_browser():
    fake_user_agent = Faker()
    options = webdriver.ChromeOptions()
    options.binary_location = '/opt/chrome/chrome'
    options.add_experimental_option("excludeSwitches", ['enable-automation'])
    options.add_argument('--disable-infobars')
    options.add_argument('--disable-extensions')
    options.add_argument('--no-first-run')
    options.add_argument('--ignore-certificate-errors')
    options.add_argument('--disable-client-side-phishing-detection')
    options.add_argument('--allow-running-insecure-content')
    options.add_argument('--disable-web-security')
    options.add_argument('--lang=' + random.choice(language_list))
    options.add_argument('--user-agent=' + fake_user_agent.user_agent())
    options.add_argument('--headless')
    options.add_argument('--no-sandbox')
    options.add_argument("--disable-gpu")
    options.add_argument("--window-size=1280x1696")
    options.add_argument("--single-process")
    options.add_argument("--disable-dev-shm-usage")
    options.add_argument("--disable-dev-tools")
    options.add_argument("--no-zygote")
    options.add_argument(f"--user-data-dir={mkdtemp()}")
    options.add_argument(f"--data-path={mkdtemp()}")
    options.add_argument(f"--disk-cache-dir={mkdtemp()}")
    options.add_argument("--remote-debugging-port=9222")
    chrome = webdriver.Chrome("/opt/chromedriver", options=options)

    return chrome

#Our main Lambda function
def lambda_handler(event, context):
    try:

        # Open browser
        browser = open_browser()

        # Clean cookies
        browser.delete_all_cookies()
        browser.set_page_load_timeout(60)

        # Open web
        # logging.info("Opening web " + os.environ['URL'])
        browser.get(os.environ['URL'])

        #get text from google
        target_XPATH = '/html/body/div[1]/div[3]/form/div[1]/div[1]/div[3]/center/input[1]'
        target = WebDriverWait(browser, 20).until(EC.presence_of_element_located((By.XPATH, target_XPATH)))
        return_val = target.text

        #browser.close() might not be required since the container is destroyed anyway after done.
        browser.close()
        return {
            "return": return_val
        }

    except AssertionError as msg:
        logging.error(msg)
        browser.close()
        sys.exit()
    except TimeoutException:
        logging.error('Request Time Out')
        browser.close()
        sys.exit()
    except WebDriverException:
        logging.error('---------------------- WebDriver-Error! --------------------', exc_info=True)
        logging.error('---------------------- WebDriver-Error! END ----------------')
        browser.close()
        sys.exit()
    except NoSuchWindowException:
        logging.error('Window is gone, somehow...- NoSuchWindowException')
```

```
        sys.exit()
    except NoSuchElementException:
        logging.error('---------------------- No such element on site. ----------------------', exc_info=True)
        logging.error('---------------------- No such element on site. END ----------------------')
        browser.close()
        sys.exit()
```

Additional requirements.

Before you push this baby out into the wild world of AWS, you need to create Amazon Elastic Container Registry (ECR) where the container will be stored. The container is 537.96 MB in size... so it's up to you if you create private repo (500 MB free) or public repo where you get 50 GB free. SAM will ask you this repo name when deploying your lambda.

Build, Test, Publish

You can test locally ->Guide<, if you set up SAM and AWS cli (look that thing up in google) and have docker installed. You can now publish it to AWS and have fun with Selenium, Chrome and Chromedriver in Lambda function.

I hope you liked this short guide and got something useful out of it. Take a break, grab some beverage and maybe one for me too.

Liked it ? Buy me a drink :)

Last update: May 26, 2022

Comments