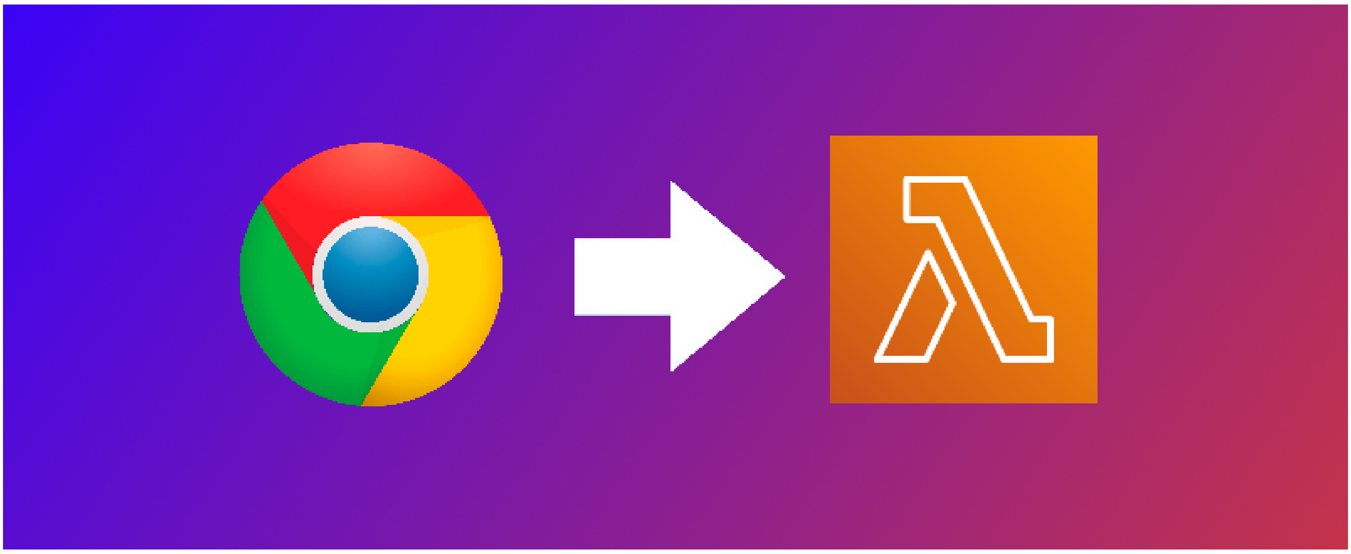
Get started Open in app

G

2 Followers About Follow

Running headful Chrome with extensions in a Lambda function

G Jan 17 · 5 min read



As a Serverless enthusiast, I tend to use Lambda for EVERYTHING. I find the combination of dev speed, scalability and integration with other AWS services irresistible and it's served me well over the last 5 years.

However, I never managed to run a fully fledged Chrome browser (with extensions) in a lambda function. Until now.

The Challenge

There are a couple issues with this use case. First, lambda by default is not made to run graphical applications. On top of that, most packages I found were either outdated (node 6.10 anyone??), and/or outright didn't run.

After a lot of unsuccessful tests, I settled for `chrome-aws-lambda` which seemed to be up to date, maintained, and working.

Hurdle #1: Getting chrome to run :

After a bit of tinkering, and a lot of googling, here are the flags I found necessary to run `chrome-aws-lambda` :

```
const chromeFlags = ['--no-xshm','--disable-dev-shm-usage','--
single-process','--no-sandbox','--no-first-run',`--load-
extension=${extensionDir}`]
```

In short, it's all about accomodating the non-standard environment and filesystem we're running on. Look here for full details: https://peter.sh/experiments/chromium-command-line-switches/

With those, I was able to run the following function :

```
const remoteInterface = require('chrome-remote-interface')
const chromium = require('chrome-aws-lambda')
const launcher = require('chrome-launcher')

const chromeFlags = [
    '--no-xshm',
    '--disable-dev-shm-usage',
    '--single-process',
    '--no-sandbox',
    '--no-first-run',
    '--window-size=1366,768',
    `--load-extension=${extensionDir}`
]

const testChrome = async event => {

    const chromePath = await chromium.executablePath
    const chromeOptions = {
        chromePath,
        chromeFlags,
        port: 9222,
        ignoreDefaultFlags: true,
        userDataDir: false,
        logLevel: 'verbose'
    }

    const chrome = await launcher.launch(chromeOptions)
    const client = await remoteInterface()
    const { Network, Page, Runtime, Console } = client

    await Network.enable()
    await Runtime.enable()
    await Console.enable()
    await Page.enable()
    await Page.navigate({ url: 'http://example.com' })
    await Page.loadEventFired()

    const res = await Runtime.evaluate({ expression: 'chrome' })
    await client.close()

    return res
};

export const handler = testChrome
```

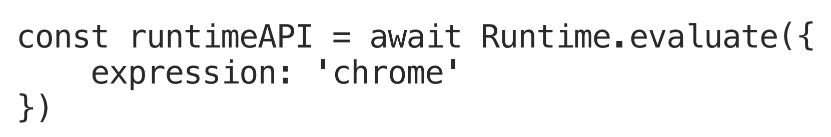
chrome.ts hosted with ❤ by GitHub view raw

Given that I was running without the `--headless` switch, I was pretty confident that I had cracked it. After a bit of cleanup and packaging all the heavy dependencies into a layer, I set out to work on talking to my extension.

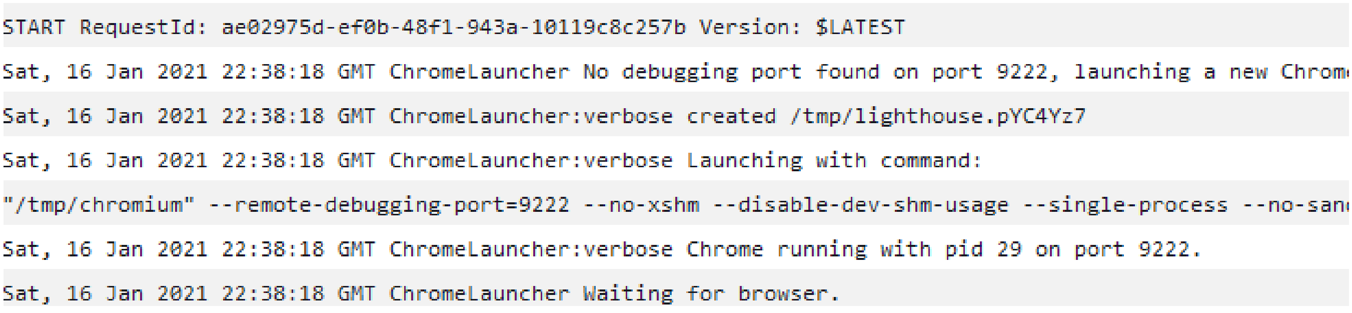
Hurdle #2: chrome.runtime :

To work with extensions, we need the `chrome.runtime` API. https://developer.chrome.com/docs/extensions/reference/runtime/

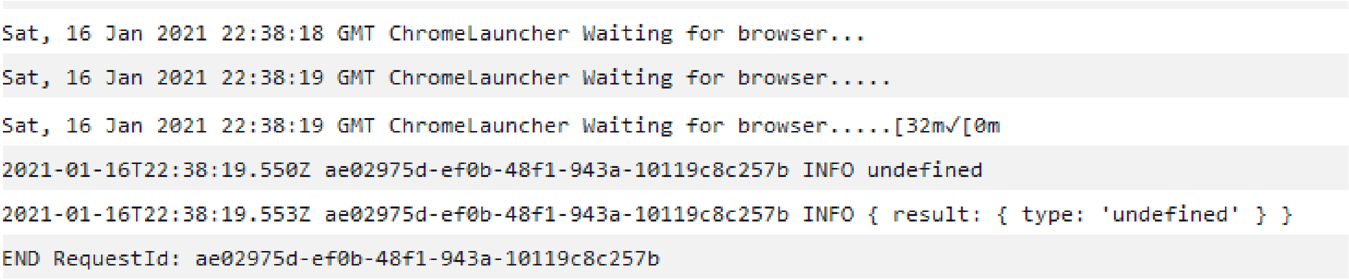
However, when running the function, the following happened :



```
const runtimeAPI = await Runtime.evaluate({
    expression: 'chrome'
})
```



```
START RequestId: ae02975d-ef0b-48f1-943a-10119c8c257b Version: $LATEST
Sat, 16 Jan 2021 22:38:18 GMT ChromeLauncher No debugging port found on port 9222, launching a new Chrom
Sat, 16 Jan 2021 22:38:18 GMT ChromeLauncher:verbose created /tmp/lighthouse.pYC4Yz7
Sat, 16 Jan 2021 22:38:18 GMT ChromeLauncher:verbose Launching with command:
"/tmp/chromium" --remote-debugging-port=9222 --no-xshm --disable-dev-shm-usage --single-process --no-san
Sat, 16 Jan 2021 22:38:18 GMT ChromeLauncher:verbose Chrome running with pid 29 on port 9222.
Sat, 16 Jan 2021 22:38:18 GMT ChromeLauncher Waiting for browser.
```



```
Sat, 16 Jan 2021 22:38:18 GMT ChromeLauncher Waiting for browser...
Sat, 16 Jan 2021 22:38:19 GMT ChromeLauncher Waiting for browser.....
Sat, 16 Jan 2021 22:38:19 GMT ChromeLauncher Waiting for browser.....[32m✓[0m
2021-01-16T22:38:19.550Z ae02975d-ef0b-48f1-943a-10119c8c257b INFO undefined
2021-01-16T22:38:19.553Z ae02975d-ef0b-48f1-943a-10119c8c257b INFO { result: { type: 'undefined' } }
END RequestId: ae02975d-ef0b-48f1-943a-10119c8c257b
```

The chrome object doesn’t exit

In my DevTools, the chrome object was definitely there. Further research led me to this very informative and disappointing post on the DevTools Github.

> Given that we don’t intend to support extensions in headless mode and there aren’t that many other useful things behind the chrome object, I don’t think implementing it is a very high priority.

copyright Disney/Pixar

Enter Container Images

When I read the news at re:Invent 2020 about container images, I felt like it was worth giving it a spin. In particular, this bit sounded very interesting :

> Lambda provides open-source runtime interface clients that you add to an alternative base image to make it compatible with Lambda.

The promise of being able to slap a Lambda runtime onto any docker image seemed almost too good to be true, and I was expecting to fail miserably (but learn stuff in the process). I needed to know.

The idea

I figured I'd need the following :

- A Docker image running `google-chrome-stable`
- The Node.js runtime interface client (RIC) from AWS

If Chrome was running in the container, `chrome-launcher` would find it and the lambda itself would stay 99% the same.

Running Chrome in Docker

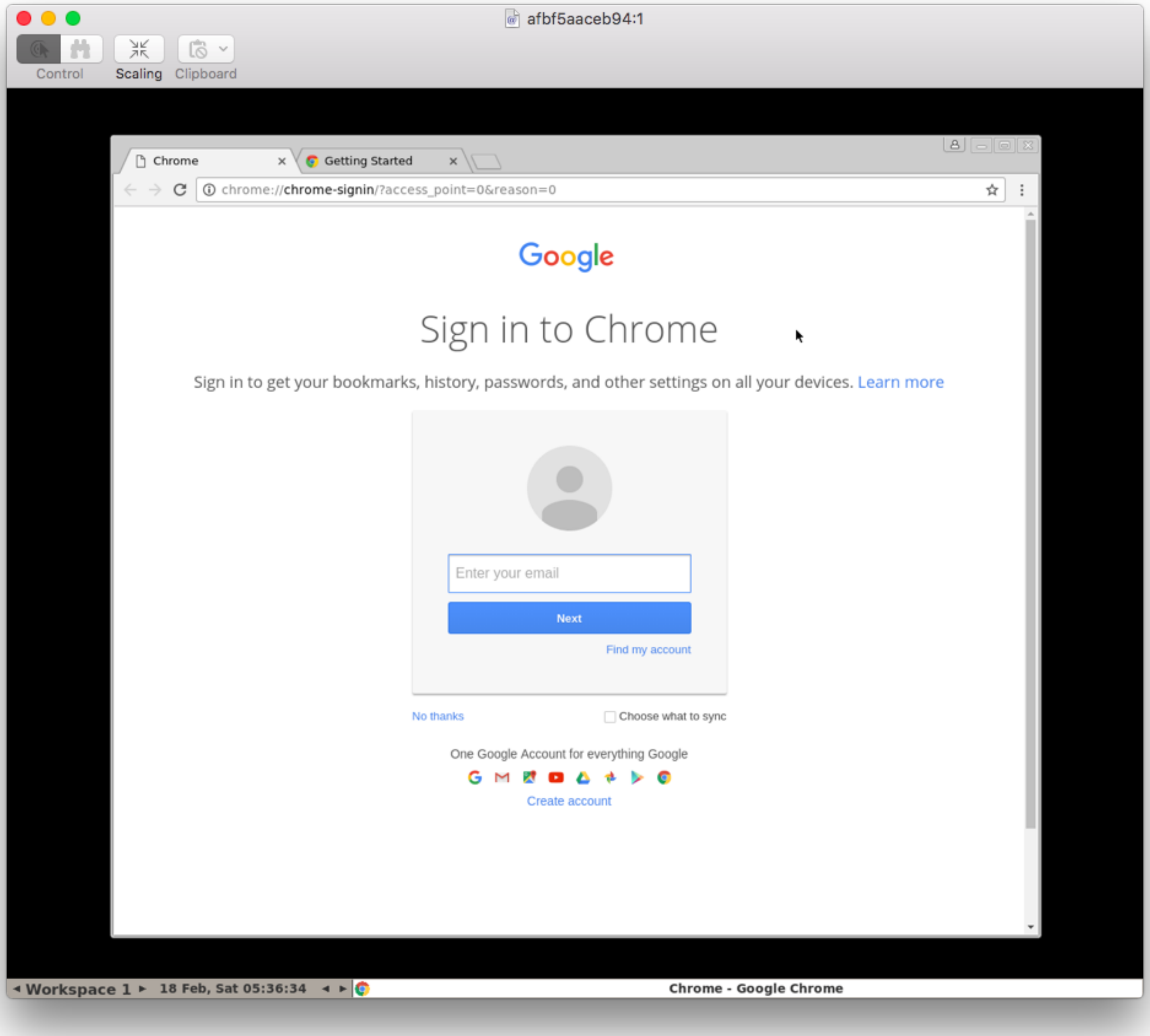
I followed this amazing guide by Stephen Fox, updated ubuntu to focal, and added `gnupg2` after an explicit error message.

```
FROM ubuntu:focal

RUN apt-get update; apt-get clean

# Add a user for running applications.
RUN useradd apps
RUN mkdir -p /home/apps && chown apps:apps /home/apps

# Install xvfb and other stuff.
RUN apt-get install -y xvfb fluxbox wget wmctrl gnupg2

# Set the Chrome repo.
RUN wget -q -O - https://dl-ssl.google.com/linux/linux_signing_key.pub | apt-key add
    && echo "deb http://dl.google.com/linux/chrome/deb/ stable main" >> /etc/apt/sou

# Install Chrome.
RUN apt-get update && apt-get -y install google-chrome-stable

COPY bootstrap.sh /
```

```
RUN chmod 755 /bootstrap.sh
ENTRYPOINT [ "/bootstrap.sh" ]
```

Dockerfile hosted with ❤ by GitHub view raw

In no time, I was able to VNC into my Docker container and launch Chrome. So far so good.



Sign in to Chrome

Sign in to get your bookmarks, history, passwords, and other settings on all your devices. Learn more

One Google Account for everything Google

Adding the Lambda RIC

You can find the `aws-lambda-ric` for Node here : https://github.com/aws/aws-lambda-nodejs-runtime-interface-client

The AWS guys have done a decent job at documenting the package and its use, it boils down to adding the following to the docker container:

```
ARG FUNCTION DIR="/function"
```

```

FROM ubuntu:focal
ARG FUNCTION_DIR

# Install utils for ric
RUN apt install -y curl wget git g++ make cmake unzip libcurl4-openssl-dev autoconf

# install NodeJS
RUN curl -sL https://deb.nodesource.com/setup_12.x | bash -
RUN apt install -y nodejs
RUN npm install -g yarn

# Install ric
RUN npm i aws-lambda-ric

ENTRYPOINT ["/usr/local/bin/npx", "aws-lambda-ric"]
```

Dockerfile hosted with ❤ by GitHub view raw

Adding the emulator

On top of the RIC, AWS provides us with a convenient Lambda Runtime Interface Emulator (RIE) to let us test our container image locally. It's available on Github, and can be added to the Dockerfile like this:

```
ADD https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/down
RUN chmod 755 /usr/local/bin/aws-lambda-rie
COPY entrypoint.sh /
RUN chmod 755 /entrypoint.sh
ENTRYPOINT [ "/entrypoint.sh" ]
```

Dockerfile hosted with ❤ by GitHub view raw

The `entrypoint.sh` script is a simple if condition that checks if we're running on lambda or locally and uses the emulator in the latter case.

```
#!/bin/sh
if [ -z "${AWS_LAMBDA_RUNTIME_API}" ]; then
  exec /usr/local/bin/aws-lambda-rie /usr/bin/npx aws-lambda-ric $1
else
  exec /usr/bin/npx aws-lambda-ric $1
fi
```

entrypoint.sh hosted with ❤ by GitHub view raw

Modifying bootstrap script for Lambda

I made a few changes to suit the task at hand :

- Removed the VNC server
- Added the emulator entrypoint

Click this for the full bootstrap file

Adding the function code

All that was left to do was adding the function code and pointing it to the app handler.

```
# App setup
RUN mkdir -p ${FUNCTION_DIR}
COPY function/package.json ${FUNCTION_DIR}
WORKDIR ${FUNCTION_DIR}

# Install deps
RUN yarn

# Build app (do last for speed)
COPY function/. ${FUNCTION_DIR}
RUN yarn build
WORKDIR ${FUNCTION_DIR}/.build

CMD ["app.handler"]
```

Dockerfile hosted with ❤ by GitHub view raw

The mods to the function itself were minimal, I removed the `chrome-aws-lambda` package and changed the flags to:

```
export const chromeFlags = ['--no-first-run','--window-size=1366,768',`--load-extension=${extensionDir}`]
```

Build and run

The build step took a little long the first time (10+ minutes), notably the ric install. Fortunately Docker caches everything and the pain was short lived

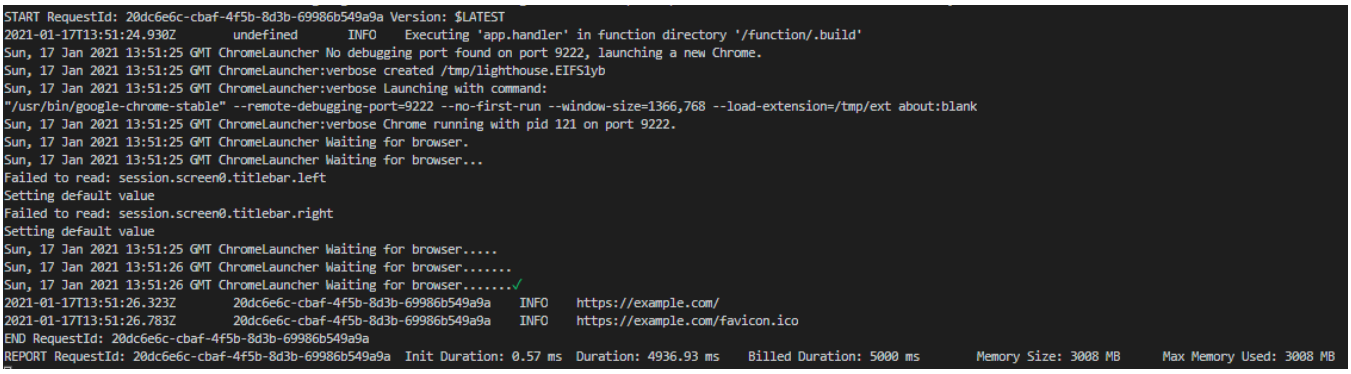
```
docker build -t chrometest:latest .
```

After that, I used the following to run the container:

```
docker run -p 9000:8080 --user apps --privileged chrometest:latest
```

And all that was left was to test. In a different terminal, I ran:

```
curl -XPOST "http://localhost:9000/2015-03-
31/functions/function/invocations" -d '{}'
```



```
START RequestId: 20dc6e6c-cbaf-4f5b-8d3b-69986b549a9a Version: $LATEST
2021-01-17T13:51:24.930Z        undefined       INFO    Executing 'app.handler' in function directory '/function/.build'
Sun, 17 Jan 2021 13:51:25 GMT ChromeLauncher No debugging port found on port 9222, launching a new Chrome.
Sun, 17 Jan 2021 13:51:25 GMT ChromeLauncher:verbose created /tmp/lighthouse.EIFS1yb
Sun, 17 Jan 2021 13:51:25 GMT ChromeLauncher:verbose Launching with command:
"/usr/bin/google-chrome-stable" --remote-debugging-port=9222 --no-first-run --window-size=1366,768 --load-extension=/tmp/ext about:blank
Sun, 17 Jan 2021 13:51:25 GMT ChromeLauncher:verbose Chrome running with pid 121 on port 9222.
Sun, 17 Jan 2021 13:51:25 GMT ChromeLauncher Waiting for browser.
Sun, 17 Jan 2021 13:51:25 GMT ChromeLauncher Waiting for browser...
Failed to read: session.screen0.titlebar.left
Setting default value
Failed to read: session.screen0.titlebar.right
Setting default value
Sun, 17 Jan 2021 13:51:25 GMT ChromeLauncher Waiting for browser.....
Sun, 17 Jan 2021 13:51:26 GMT ChromeLauncher Waiting for browser.......
Sun, 17 Jan 2021 13:51:26 GMT ChromeLauncher Waiting for browser.......✓
2021-01-17T13:51:26.323Z        20dc6e6c-cbaf-4f5b-8d3b-69986b549a9a    INFO    https://example.com/
2021-01-17T13:51:26.783Z        20dc6e6c-cbaf-4f5b-8d3b-69986b549a9a    INFO    https://example.com/favicon.ico
END RequestId: 20dc6e6c-cbaf-4f5b-8d3b-69986b549a9a
REPORT RequestId: 20dc6e6c-cbaf-4f5b-8d3b-69986b549a9a  Init Duration: 0.57 ms  Duration: 4936.93 ms    Billed Duration: 5000 ms        Memory Size: 3008 MB    Max Memory Used: 3008 MB
```

container log

Seeing the familiar Lambda START - END logs in my container felt very satisfying, but the result of the function itself made me jump up and down like a madman:

```
g@desktop:/mnt/c/Node/chrometest$ curl -XPOST "http://localhost:9000/2015-03-31/funct

{
  "result":{
    "type":"object",
    "className":"Object",
    "description":"Object",
    "objectId":"{\"injectedScriptId\":2,\"id\":1}"}
}
```

log.json hosted with ❤ by GitHub view raw

FINALLY! The chrome object is defined, we have a fully fledged “headful” chrome browser, on a lambda runtime, able to use the `chrome.runtime` API.

copyright Disney/Pixar

All that’s left to do is dump the container on AWS, and voila!

Ship to AWS

To use the container in your lambda functions, just deploy it to the ECR registry as described here

```
aws ecr get-login-password --region us-east-1 | docker login --
username AWS --password-stdin 123456789012.dkr.ecr.us-east-
1.amazonaws.com

docker tag  chrometest:latest 123456789012.dkr.ecr.us-east-
1.amazonaws.com/chrometest:latest
docker push 123456789012.dkr.ecr.us-east-
1.amazonaws.com/chrometest:latest
```

Would I use again?

Frankly, I don’t think I’ll have that many use cases for Lambda Container Images. That being said, it is a great addon to AWS suite and fills a gap in the Serverless ecosystem that was missing for years. I wasn’t expecting it to be that painless, the whole thing took a few hours, and it “just worked”.

Full project code on Github

Chrome AWS Lambda Containers Docker