Aldira Putra R

Testing in Infrastructure as Code

Test in Infrastructure as Code

Aldira Putra Raharja

This book is for sale at http://leanpub.com/test-iac

This version was published on 2023-12-12

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2023 Aldira Putra Raharja

Contents

Preface **1**
Scope 1

Container Unit Test **3**
Container Structure Test – (CST) 3
Sample Use Cases 5

Kubernetes Manifest Unit Test **13**
Test using dry run 13

Preface

Infrastructure as code – or IaC are a great way to handle operations especially in Cloud Native era. By declaring our Infrastructure as a code, enables many great use cases like automation, code reviews, even testing.

A lot of tools like Terraform, Ansible, Helm, mainly focus on automation. Simply write a simple code that declare a `resource` with all the necessary configuration, run it in some form of automation, and voilà, your cloud infrastructure has been provisioned.

But in reality, there are more than just provisioning a new resource. What about updating those configurations? Is our resources are configured properly? Can we make sure that our configuration is correct? Do our `resources` properly refer to one and another?

With IaC, those questions should be answered easily. Those are *code* after all, right? When learning a new programming language, there always a chapter talking about unit test – how to test the code before it runs on production.

Unfortunately, not many resources actually talk about how to do test in IaC. Many resources mainly talk about how to write IaC code, run the code, how to roll back the changes, but not on how to run the test.

That's the main reason why I wrote this book. I believe that we should also treat our IaC as a real code. A code that testable to avoid unnecessary downtime due to misconfiguration.

Scope

This book mainly focusing on testing IaC in the realm of Kubernetes, Helm, Docker, and some other IaC like Terraform or Pulumi. Each chapter will consist of an explanation of why we need to do test, how to do tests, and some use cases.

Container Unit Test

Cloud native infrastructure like Kubernetes mainly use container to run the application. A common flow to ship the application is to build it as a container image with all the dependencies, both direct dependencies like a binary, or operating system library. There is also an occasion that application require some specific file to be present inside the container.

But then, how to validate those dependencies really present inside the container? Do the runtime *USER* has required permission to access those? Do the dependencies has correct version? These often become the main topic of testing a container. To answer those questions, there is a tool called Container Structure Test.

Container Structure Test – (CST)

> https://opensource.googleblog.com/2018/01/container-structure-tests-unit-tests.html

Container Structure Test is a tool to test container to ensure the structure is correct, reducing potential runtime issues. Referring directly from the blog.

> The Container Structure Tests provide a powerful framework to validate the structure of a container image. These tests can be used to check the output of commands in an image, as well as verify metadata and contents of the file system.

There are various type of testing that CST support like `commandTest`, `fileExistenceTests`, and many more[1]. Test configuration is written in YAML or JSON format. Consider the following example.

```yaml
schemaVersion: 2.0.0

commandTests:
  # Checking that the output of the image has node versio\
n of 20
  - name: node version
    command: node
    args:
      - -v
    expectedOutput:
      - 20.0.0
```

Runs the test against the official node 20 will succeed

```
 container-structure-test test -c node-test.yaml -i node\
:20 --pull
20: Pulling from library/node
Digest: sha256:445acd9b2ef7e9de665424053bf95652e0b8995ef3\
6500557d48faf29300170a
Status: Image is up to date for node:20

===================================
====== Test file: node-test.yaml ======
===================================
=== RUN: Command Test: node version
--- PASS
duration: 119.008875ms
stdout: v20.10.0
```

[1] https://github.com/GoogleContainerTools/container-structure-test

```
=====================================
=============== RESULTS ===============
=====================================
Passes:      1
Failures:    0
Duration:    119.008875ms
Total tests: 1

PASS
```

Sample Use Cases

Verify Dependency Version

Let's say a QA Team runs their UI Automation Test using Selenium on a Kubernetes cluster, and they need to manage their own Selenium Node Image on their own. They are using the official Dockerfile from the main repo[2] as follows

```
ARG NAMESPACE
ARG VERSION
ARG AUTHORS
FROM ${NAMESPACE}/node-base:${VERSION}
LABEL authors=${AUTHORS}

USER root

#==========================================
# Google Chrome
#==========================================
ARG CHROME_VERSION="google-chrome-stable"
RUN wget -q -O - https://dl-ssl.google.com/linux/linux_si\
```

[2] https://github.com/SeleniumHQ/docker-selenium/blob/trunk/NodeChrome/Dockerfile

```
gning_key.pub | gpg --dearmor | tee /etc/apt/trusted.gpg.
d/google.gpg >/dev/null \
  && echo "deb http://dl.google.com/linux/chrome/deb/ sta\
ble main" >> /etc/apt/sources.list.d/google-chrome.list \
  && apt-get update -qqy \
  && apt-get -qqy install \
    ${CHROME_VERSION:-google-chrome-stable} \
  && rm /etc/apt/sources.list.d/google-chrome.list \
  && rm -rf /var/lib/apt/lists/* /var/cache/apt/*

...

USER 1200

#=========================================
# Chrome webdriver
#=========================================
ARG CHROME_DRIVER_VERSION
RUN if [ ! -z "$CHROME_DRIVER_VERSION" ]; \
  then CHROME_DRIVER_URL=https://edgedl.me.gvt1.com/edged\
l/chrome/chrome-for-testing/$CHROME_DRIVER_VERSION/linux6
4/chromedriver-linux64.zip ; \
  else echo "Geting ChromeDriver latest version from http\
s://googlechromelabs.github.io/chrome-for-testing/LATEST_
RELEASE_" \
    ...

...
```

The Dockerfile specify how to install Google Chrome and Chrome webdriver, as well as other related configuration.

Its seems very straightforward at first, but there's a hidden, crucial details about this Dockerfile. Google Chrome and Chrome webdriver must be [compati-

ble](https://chromedriver.chromium.org/downloads[3] to work properly. From the example above, we can specify the chrome driver version via `CHROME_DRIVER_VERSION` arguments, but we can't specify the Google Chrome version. So, even if the build is succeeded, there's a potential issue on the runtime, and the Selenium Test could be failed to run.

To prevent that issue, CST can be used for testing the image to verify whether Google Chrome version is compatible with our desired chrome driver.

`expectedOutput` can be used to verify parts of the output. This capability greatly simplifies the assertion.

```yaml
schemaVersion: 2.0.0

commandTests:
  # Verifying the binary to be present in the container
  - name: Verify Google Chrome binary is available in path
    command: google-chrome
    args:
      - --version
    expectedOutput:
      - Google Chrome
  - name: Verify Chrome Driver is available in path
    command: chromedriver
    args:
      - --version
    expectedOutput:
      - ChromeDriver
  # Verifying the major version
  - name: Google Chrome Major Version
    command: bash
```

[3] https://chromedriver.chromium.org/downloads/version-selection

```
    args:
      - -c
      - google-chrome --version | sed -E "s/.* ([0-9]+)(\\
.[0-9]+){3}.*/\1/"
    expectedOutput:
      - 120
  - name: Chrome webdriver version
    command: bash
    args:
      - -c
      - chromedriver --version | sed -E "s/.* ([0-9]+)(\.\
[0-9]+){3}.*/\1/"
    expectedOutput:
      - 120
```

When executed against the image, the output will be similar with the following

```
 container-structure-test test -c selenium-node-chrome.y\
aml -i selenium/node-chrome --pull --platform linux/amd64
latest: Pulling from selenium/node-chrome
Digest: sha256:3b59c85337e7a1bcbc2b2068caed9187505d7a68cc\
f0ec3c9bb698e6b3a9972e
Status: Image is up to date for selenium/node-chrome:late\
st

===================================================
====== Test file: selenium-node-chrome.yaml ======
===================================================
=== RUN: Command Test: Verify Google Chrome binary is ava\
ilable in path
--- PASS
duration: 272.861042ms
stdout: Google Chrome 120.0.6099.62
```

```
=== RUN: Command Test: Verify Chrome Driver is available \
in path
--- PASS
duration: 128.315041ms
stdout: ChromeDriver 120.0.6099.62 (93ab19d804d479b507c46\
94305631dfb65f8e930-refs/branch-heads/6099_56@{#3})

=== RUN: Command Test: Google Chrome Major Version
--- PASS
duration: 289.275584ms
stdout: 120

=== RUN: Command Test: Chrome webdriver version
--- PASS
duration: 156.515875ms
stdout: 120


==================================================
===================== RESULTS ====================
==================================================
Passes:      4
Failures:    0
Duration:    846.967542ms
Total tests: 4

PASS
```

Simple Security Testing

When building a container, some files might contain sensitive data. It is highly recommended to put those files names or path into .dockerignore file. To verify that the container doesn't contain sensitive data, `fileExistenceTests` and `fileContentTests` can be

used.

For example, here are the Dockerfile to build a very simple NodeJS application

```
FROM node:20

WORKDIR /home/node/app

COPY . .

RUN npm ci

ENTRYPOINT [ "npm", "start" ]
```

From that Dockerfile, it is clear that `/home/node/app` might contain more files than it should due to `COPY . .` command. Here are example of tests to validate sensitive files not added to the container

```yaml
schemaVersion: 2.0.0

commandTests:
  # Checking that the output of the image has node versio\
n of 20
  - name: node version
    command: node
    args:
      - -v
    expectedOutput:
      - v20

fileExistenceTests:
  - name: .env
    path: /home/node/app/.env
    shouldExist: false

```

```
fileContentTests:
  - name: .npmrc env var
    path: /home/node/app/.npmrc
    # This example mainly checks that the .npmrc uses ARG\
, not a hard coded token.
    expectedContents:
      - //registry.npmjs.org/:_authToken=${NPM_TOKEN}
```

And here's the sample output when it detects those files

```
container-structure-test test -c security-test.yaml -i \
test-iac/node-usecase:1

===========================================
====== Test file: security-test.yaml ======
===========================================
=== RUN: Command Test: node version
--- PASS
duration: 128.956166ms
stdout: v20.10.0

=== RUN: File Content Test: .npmrc env var
--- FAIL
duration: 0s
Error: Expected string '//registry.npmjs.org/:_authToken=\
${NPM_TOKEN}' not found in file content string '//registr
y.npmjs.org/:_authToken=oops-my-token-is-exposed'
=== RUN: File Existence Test: .env
--- FAIL
duration: 0s
Error: File /home/node/app/.env should not exist but does

===========================================
================ RESULTS ==================
===========================================
```

```
Passes:      1
Failures:    2
Duration:    128.956166ms
Total tests: 3

FAIL
FATA[0000] FAIL
```

Kubernetes Manifest Unit Test

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-iac

Test using dry run

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-iac

Use Cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-iac

Validate manifest format

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-iac

Validate manifest against Validation Admission Webhook

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/test-iac