Using containers for reproducible results

A talk on why and how we use containers at ECMWF to achieve a certain degree of reproducibility on some of out web-based services.

This presentation was given at the *Building reproducible workflows for earth sciences* (https://www.ecmwf.int/en/learning/workshops/building-reproducible-workflows) workshop, and is based on the work of Marcos Hermida, Manuel Martins and Carlos Valiente.

The original Markdown source for this document is available at https://github.com/carletes/using-containers-for-reproducible-results

Motivation

We run several web services at ECMWF, mostly developed in-house. An example is the EFAS website (https://www.efas.eu/efas_frontend/#/home).

We need to know what we're running on our servers: What precise version of our code, *and* the runtime it relies on.

We keep all our code under Git. We need to know how how to get from a given version of our code to a deployed state in our servers.

We have a relatively short release cycle (weeks), and we rarely roll back or abort releases. Our reproducibility requirements are, therefore, very short-term.

Reproducible runtime

Containers let us bundle together our code with all its dependencies:

```
# Dependency: Python run-time.
FROM python

# Dependency: Some Python third-party modules
RUN set -eux \
    && pip install numpy

# Our code.
COPY ./myapp.py /code/myapp.py
```

In our build pipeline we execute commands similar tho this one:

```
$ docker build \
    -t myapp:0.1 \
    -f Dockerfile-0.1 \
    --build-arg http_proxy=$http_proxy \
    --build-arg https_proxy=$https_proxy \
    .
```

That trivial example shows that the Python runtime we get is whatever happens to be the default offering on the Docker Hub.

```
$ docker run --rm myapp:0.1 python --version
Python 3.7.4
$
```

That is not what we want:

- Even minor versions of the Python run-time may introduce incompatible changes …
- … and so do libraries and utilities from the base operating system image .

So we do something like this instead:

```
FROM python:3.7.1-stretch

RUN set -eux \
    && pip install numpy==1.17.1

COPY ./myapp.py /code/myapp.py
```

This gives us something more reproducible, as far as our dependencies are concerned:

```
$ docker build \
    -t myapp:0.2 \
    -f Dockerfile-0.2 \
    --build-arg http_proxy=$http_proxy \
    --build-arg https_proxy=$https_proxy \
    .

$ docker run --rm myapp:0.2 python --version
Python 3.7.1
$
```

Linking the runtime to its source

We have adopted the *convention* of tagging our container images with similar tags to those we use for our Git repositories:

Git branch/tag	Container tag
`develop`	`latest`
`feature/*`	`feature-*`
`release/yyyy-mm-dd-xxx`	`yyyy-mm-dd-xxx-rc`

Git branch/tag	Container tag
`yyyy-mm-dd-xxx`	`yyyy-mm-dd-xxx`

This is good enough for us, but, since *both Git tags and container image tags are moveable*, perhaps we should be doing something else — embedding the Git repository hash inside the container images themselves:

```
FROM python:3.7.1-stretch

RUN set -eux \
    && pip install numpy==1.17.1

COPY ./myapp.py /code/myapp.py

ARG git_hash="<unknown>"
LABEL git_hash=$git_hash
```

Images built like this inequivocably refer to the source code used to build them:

```
$ docker build \
    -t myapp:0.3 \
    -f Dockerfile-0.3 \
    --build-arg http_proxy=$http_proxy \
    --build-arg https_proxy=$https_proxy \
    --build-arg git_hash=$(git rev-parse HEAD) \
    .

$ docker inspect myapp:0.3 | jq '.[].ContainerConfig.Labels'
{
  "git_hash": "532aa0fe568c79c59e5cececc18732b2a3cf8995"
}
$
```

Deploying known versions

We run the EFAS website on Kubernetes (https://kubernetes.io/). We use deployment descriptors similar to this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myapp:0.3
```

Those deployment descriptors are kept in a Git repository, which is also tagged with the name of the release they refer to.

Very much like in the case of individual images, perhaps we should use the image hashes instead of their image tags, since hashes are immutable:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myapp@sha256:22b0e758ea05cb20f4ac5d18de5869152054e05cedecb3b46cbfc839101423ef
```

Summary: Partial reproducibility

So far we got:

- A way to identify the binary images we want to run.
- A way to identify the source code used to build those images.
- A way to identify the versions of *some* of our third-party dependencies.

Our current workflow assumes that, whenever we want to build our container images,

- Several public repositories (the Docker Hub container image registry and the Python Package Index) are available whenever we want to build our container images.

- The particular versions of the packages in the bottom-most images (C library, system components, compilers, . . .) are reasonably stable during our release cycle.

If you need stronger reproducibility, you could consider the following options:

- Mirror the public repositories you rely on.
- Consider things like distroless images (https://github.com/GoogleContainerTools/distroless) as your base images. This approach gives you precise versions of all your binary dependencies.
- Consider build systems like Bazel (https://bazel.build/) and its support for building container images (https://github.com/bazelbuild/rules_docker) for building your images, which aim for deterministic builds.