SPLAB

transcript

type:
tutorial

distribution:
public

status:
final

initiative:
Cloud-Native
Applications

Companion Guide/Transcript to 'Acquisition and Exploitation of Insights into Cloud Applications' – CCEM 2020

Josef Spillner and Panagiotis Gkikopoulos
Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/splab/)
8401 Winterthur, Switzerland
{josef.spillner,pang}@zhaw.ch

November 18, 2021

UPDATED November 2021

Learning Objective

The Microservice Artefact Observatory (MAO) is a long-term collaborative effort with global observation sites to assess, through static and dynamic analysis of metadata, code, configuration and runtime behaviour, the characteristics of cloud software artefacts. This tutorial explains how such insights can be obtained and what the acquired data are good for when attempting to overcome identified consistency, quality or security issues.

Note: This transcript contains ready-to-repeat commands for the first successful steps with MAO, Kubernetes, OpenShift and Docker and other cloud and data science technologies based on pre-provisioned virtual machines. The appendix contains code to reproduce the machines, as well as the configuration of a sample application. The tutorial is meant to convey practical research results. For in-depth tutorials without the research perspective, please refer to the Kubernetes Tutorials section at `https://kubernetes.io/docs/tutorials/`. Specifically for OpenShift, there is also the excellent APPUiO Techlab (in German) at `https://github.com/appuio/techlab`.

Note: This tutorial transcript is partly based on previous ones including 'Quality and Feedback Techniques in Kubernetes Application Engineering' given at IC2E 2019, with revisions for 'Data-Supported Quality Analysis of Microservice Artefacts in Software Development' given at the CH Open Workshop Days 2019, and 'Microservices in Numbers: Diagnostic Docker Deep Dive' given at CLOSER 2020.

Note: Copying and pasting commands directly from the PDF may fail due to the insertion of spaces. It is advised to learn the commands by re-typing them instead of copying them.

Contents

1 Application Deployment **2**

2 Application Management **3**

3 Customisation and Packaging **3**

4 Dockerfile Quality Assessment **4**

5 Working with Docker Images **5**

6 Basic Helm Chart Quality Assessment **6**

7 Advanced Helm Chart Quality Assessment **7**

8 Label Consistency Checks **7**

9 Lambda Functions **8**

10 Kubernetes Operators **9**

11 DApps **9**

12 Mixed-Technology Application **9**

13 MAO Tools **9**

14 Appendices A–C **11**

1 Application Deployment

This example explains how to deploy and invoke a simple test service on Kubernetes.

Create a file 'whale.yaml' with the contents from Appendix B. The file contains a deployment object to deploy a container, a service object to expose the service offered by the container to the outside world, and a namespace object to ensure that the application becomes manageable through a single namespace. The first command from Listing 1 deploys the application. The second one shows the deployed objects including runtime-generated information across all namespaces. Namespacing can be restricted with `-namespace closer2020-whale-namespace`; if not specified, only the default namespace (`default`) is exposed. With the `wide` option, additional deployment attributes are shown. The third command invokes the service at its backend pod with the original (unmapped) port number exposed by the Docker container image. Note that the IP address can differ, as informed by the prior command.

Listing 1: Sample service deployment

```
# first, deploy entire configuration consisting of three
    objects
kubectl create -f whale.yaml
# verify deployment status
kubectl get -A all # -o wide
# access the web service
lynx -dump http://10.1.1.3:8000
```

Other output options are possible, including `-o json` and `-o yaml` for machine-readable structured output. Note that even with the `all` meta-object specified, some internal objects are not shown, and the output format may not be readable; there are plugins to overcome these limitations.

2 Application Management

Once deployed, an application can be managed to accomodate its workload and user access patterns. By default, each deployment runs as singular instances without replicas – technically, the number of replicas is 1. Through static scaling or dynamic autoscaling, the number of replicas can be adjusted.

The `kubctl scale` command is used to scale replicas statically to a desired count, subject to availability of resources. Listing 2 shows how to scale the scenario service.

Listing 2: Sample service scaling

```
# first, set desired scaling factor
kubectl scale deployment.apps/closer2020-whale-deployment --
    replicas=3
# note: deployment.apps and deployments are identical terms
    in this case
kubectl get deployments
# repeat multiple times until n out of n are ready
kubectl get pods
kubectl get pods
# should show multiple pods now; deleting one should lead to
     restore through self-healing
kubectl delete ‘kubectl get pods | head -2 | tail -1 | cut -
    d " " - f 1‘
kubectl get pods
```

Autoscaling defines the desired number of replicas based on CPU load or custom metrics.

3 Customisation and Packaging

Many tools and formats exist to make Kubernetes applications more manageable: Kustomize, Helm, and operators are covered in this tutorial; Kedge, Kap-

itan, Draft and Flux are further popular tools.

Listing 3 demonstrates how to render configuration objects from a Helm chart and how to deploy the Helm chart either directly or through the rendered configuration.

Listing 3: Using Helm

```
git clone https://github.com/gomods/athens
cd athens/charts/athens-proxy/
helm template . > rendered.yml
kubectl create -f rendered.yml
# alternative: helm install . (requires tiller extension)
```

Listing 4 shows how to modify an application consistently across all configuration objects. In this case, a pre-defined application is deployed as development flavour alongside the production version, for instance to see behavioural differences directly within the same Kubernetes cluster. The listing furthermore hints at how to use Kustomize for overlays in case the built-in patching rules are not sufficient.

Listing 4: Using Kustomize

```
mkdir -p base overlays/newfeature
cp ~/whale.yaml base/
cat > base/kustomization.yaml << EOF
> nameSuffix: -development
> resources:
>   - whale.yaml
> EOF
kubectl apply -k base
# ... fill overlays/newfeature/kustomize.yml
kustomize build overlays/newfeature/
```

4 Dockerfile Quality Assessment

Dockerfiles may range from simple examples with 4-5 lines to more complex custom setups. It is often difficult to manually diagnose quality issues, as some are less obvious than others and may result on long build times and inefficient use of resources. It is thus a good idea to use a linter for Dockerfiles before running a build, though such a tool is not standard in Docker. In this example, hadolint will be used (Listing 5), though alternatives exist such as dockerlint and dockerfile_lint.

Listing 5: Running Hadolint

```
# Download the sample Dockerfile
# Short link for convenience; otherwise: wget https://gist.
    githubusercontent.com/EcePanos/
    efee3c71f501a40e302b8318ae0e9c76/raw/Dockerfile
```

```
wget http://bit.ly/dockersample

# Pipe it into the linter to get info
docker run --rm -i hadolint/hadolint < Dockerfile
```

Now we can start fixing the mistakes one by one. The Dockerfile in Listing 6 is a compilation of bad practices and anti-patterns, meaning there are numerous errors.

For now, feel free to delete all ENTRYPOINT instructions except the last one to clear some of the errors.

To fix the DL3018 and DL3019 warnings, pin the exact version of the package to install and add the –no-cache switch to apk add commands

Another simple optimization is to use COPY and not ADD to copy files and directories. ADD is a more advanced instruction that will download remote files and extract archives.

Listing 6: Dockerfile fixes

```
# To fix DL3025 refactor the last ENTRYPOINT command like so
    to format the arguments in JSON:
ENTRYPOINT ["nginx", "-g daemon off"]

# To fix DL4006 add the following line before RUN commands
   with pipes:
SHELL ["/bin/bash", "-o", "pipefail", "-c"]
```

It would be benefitial to get a combined (and cleansed) reporting from all of the Dockerfile linters. The 'DQA' (Dockerfile Quality Analysis) tool does exactly that. Invocation:

Listing 7: DQA invocation

```
docker run --rm -i -v /var/run/docker.sock:/var/run/docker.
   sock svl7/dqa-mao:0.1
```

5 Working with Docker Images

Docker container images differ in quality depending on how they were built and how adaptive and descriptive they are when they run. The example in Listing 8 compares the execution of two images. One of them has specific hardware requirements which on most systems will not be set up correctly so that the execution will fail.

Listing 8: Running a Docker image requiring SGX and a simple one for comparison

```
docker pull curlimages/curl:latest
docker pull oeciteam/sgx-test
```

```
docker run -ti curlimages/curl 'https://api.exchangeratesapi
    .io/latest?base=CHF&symbols=CZK'
docker run -ti oeciteam/sgx-test
```

The corresponding error message looks like the following in Listing 9.

Listing 9: Typical error message when SGX is not present

```
oe_create_helloworld_enclave(): result=21 (OE_PLATFORM_ERROR
    )
```

The underlying issues are that first, hardware requirements are not reflected in the images metadata beyond the CPU architecture and operating system, and second, containers adhere to a binary distinction of being able to run or not, without support for adaptivity and controlled graceful degradation. Further quality differences appear when trying to contact the person responsible for the image. In one case, the best practices to set maintainer data is followed, in another case not (Listing 10).

Listing 10: Inspection of both Docker images

```
docker inspect curlimages/curl:latest | jq ".[0].Config.
    Labels"
docker inspect oeciteam/sgx-test | jq ".[0].Config.Labels"
```

Quality issues encompass security and up-to-dateness aspects that warrant further drill-down into the image contents. You can learn more about image and layer analysis using the `inheritancetree.sh` script [5].

6 Basic Helm Chart Quality Assessment

Helm includes its own linter for checking your Charts. This example (Listing 11) will demonstrate its basic capabilities.

Listing 11: Helm linting

```
# Create a blank chart
helm create test
cd test
# Try to lint it
helm lint
```

No failures were found though it recommends the addition of an icon (Listing 12).

Listing 12: First chart improvement

```
# Add this line at the bottom of Chart.yaml
icon: image.png
```

This time there is an error. The icon field is not a valid URL (Listing 13).

Listing 13: Second chart improvement

```
# Change the line:
icon: http://localhost/image.png
```

The error was fixed. Notice how missing fields don't trigger the linter. Let's try something more complex, like adding maintainers (Listing 14).

Listing 14: Third chart improvement

```
# Add this line:
maintainers: foo
```

This error message is harder to read. What it means is the maintainers field requires a list (Listing 15).

Listing 15: Fourth chart improvement

```
# Modify to this:
maintainers:
  - name: foo
```

Other attributes can be added to each maintainer item, such as email and URL. The linter will inform us if we format it wrong.

7 Advanced Helm Chart Quality Assessment

Helm charts may contain several quality issues, such as omission of recommended metadata, inconsistencies, under-utilisation of templating to cut down duplicate values, references to obsolete dependencies and so forth. Very few are caught by the built-in linting mechanism of Helm. Several more, albeit presently not all, are caught by HelmQA, a versatile tool running on the command line, as Docker container, or as web server to check the quality of one or more charts. For more information, read [6].

Listing 16 shows the basic use of HelmQA on the command-line with a local invocation as application container.

Listing 16: Assessing Helm chart

```
git clone https://github.com/gomods/athens
cd athens/charts/athens-proxy/
helm lint
# will output: 1 chart(s) linted, no failures
docker run -ti -v $PWD:/charts jszhaw/helmqa
# check for Anomalies line...
```

8 Label Consistency Checks

This tool reports on the coverage and consistency of labels on Kubernetes objects. (It works as well for OpenShift templates and Docker Compose files.)

Listing 17 shows a basic usage with one OpenShift configuration file referencing two `ImageStream` objects.

Listing 17: Assessing Kubernetes configurations

```
git clone https://github.com/serviceprototypinglab/label-
    consistency
cd label-consistency
python3 labelchecker.py -u 'https://github.com/appuio/baas/
    blob/dev/deployment/imagestream.yaml'
# add [-e <kafka>/<space>/<series>] to track evolution of
    coverage over time
```

You can also use the hosted (and extended) version of the tool specifically aimed at Docker Compose files called DCValidator [1]. Access it here: `http://160.85.252.231:3000/`

9 Lambda Functions

This set of scripts assesses the state of SAM applications and associated Lambda functions [7]. For assessing the SAMs offered through SAR, pre-produced data is available without the need to run the scripts (Listings 18, 19, 20).

Listing 18: Inspecting the dataset

```
# have a look at pre-produced data
rsync 160.85.252.44::
# clone the pre-produced data (ca. 650 MB in September 2019)
rsync -avz 160.85.252.44::sams mysams
```

Listing 19: Reproducing the dataset

```
# get code and list of SAMs from SAR
git clone https://github.com/serviceprototypinglab/aws-sar-
    analysis
cd aws-sar-analysis
python3 autocontents.py --custom
# verify autostats.csv & autocontents-YYYY-MM-DD.csv
mkdir autostats; ln -s ../autocontents-2019-09-10.csv
    autostats/autocontents-2019-09-10.csv
# modify autostats/autocontents-2019-09-10.csv to contain
    only the 20 first lines
# get SAM implementations
python3 codechecker.py
# produce valid SAM folder structure, in analogy to the
    above
python3 samfinder/samfinder.py --copycode
```

Listing 20: Assessing SAMs

```
python3 samfinder/samanalyser.py _sams/
python3 insights.py all
python3 dupes/dupedetector.py autocontents-*.csv
```

10 Kubernetes Operators

... ad-hoc only!

11 DApps

... ad-hoc only! For more information, read [4].

12 Mixed-Technology Application

We have a look at Composeless, a tool which runs enhanced docker-compose files with references to cloud functions. Both containers and functions are retrieved from hubs if not present locally. Functions are executed through the Snafu engine which is auto-loaded if not present locally (Listing 21).

Listing 21: First steps with Composeless

```
git clone https://github.com/serviceprototypinglab/
    composeless
cd composeless
./composeless
```

13 MAO Tools

The MAO orchestrators was reworked significantly in 2021, including documentation improvements. Please refer to the up-to-date documentation in the orchestrator's repository instead of following the guide below. Access: ***https://github.com/serviceprototypinglab/mao-orchestrator***

These tools are emerging out of the collaborative effort on a Microservices Artefact Observatory. Instead of performing ad-hoc invocations of assessment tools for individual technologies, leading to low reproducibility and comparability, MAO is a decentralised architecture for regular tracking of the quality and evolution characteristics of software artefacts, in particular around microservices [3, 2].

This tutorial covers a simplified installation of the MAO Orchestrator. It will set it up on a single node with no distributed features enabled. As prerequisites, ensure the following features are installed on the machine you will use for this tutorial: Docker, Docker Compose, Python 3.6+.

Clone the github repository `https://github.com/serviceprototypinglab/mao-orchestrator`. Open the `docker-compose.yml` file and edit the following line in the `mao1` service, as shown in Listing 22.

Listing 22: MAO configuration

```
volumes:
      # Mount the host's Docker socket
      - /var/run/docker.sock:/var/run/docker.sock
      # Mount the data directories from the config file
      - /home/panos/tutorial:/data
      - /home/panos/audit_temp:/audit_temp
```

Change the data path to the directory you want the data files to be downloaded. Delete the lines that contain `audit_temp` in the same service as the distributed attestation feature will be disabled. Finally, run `docker-compose up -d` to setup the Orchestrator.

To simplify interactions with the Orchestrator, you can use `maoctl.py` as a CLI client. For this tutorial, we will register a tool and run it once. At the moment, most MAO assessment tools are not yet integrated with the orchestrator, but the following are already available for collecting metrics:

- dockerhub-collector: Scrapes the public DockerHub API for image metadata. Takes 15 minutes to execute.
- dqa: Scans for local repositories containing Dockerfiles.
- artifacthub: Accesses the ArtifactHub API to retrieve Helm charts and other cloud artefacts.

Moreover, there are two pseudo tools available for tests and simulations.

- maomock: Has a snapshot of the Dockerhub-collector's data saved and rebrands it with today's data. Used for testing.
- spike: It generates a set of fake data snapshots that contain a spike to trigger the spike detector.

In the real system, any tool registered by a member of the federation is visible to all others. However, your node is isolated, so we need to register the tool from scratch. The following command can be used to register the Dockerhub-collector and to run the tool immediately once (Listing 23).

Listing 23: MAO configuration

```
python3 maoctl.py tool add dockerhub-collector Panos
    panosece/dockerhub-collector:v1 https://github.com/
    EcePanos/Dockerhub-Data.git https://github.com/EcePanos/
    Dockerhub-Collector.git Docker

#Arguments :
```

```
#- tool add: A command for interacting with the 'POST /
    registry/tools' endpoint.
#- Name of the tool. You will use this name to invoke the
    tool.
#- Author (informative entry only, not used)
#- Public docker image to be used to run the tool
#- Data repository (will be cloned the first time the tool
    runs, is mounted as the data volume to the tool's
    container)
#- Code repository for the tool's source (informative entry
    only, not used)
#- Artefact this tool is concerned with (informative entry
    only, not used)

python maoctl.py tool run dockerhub-collector
```

You will see an assessment report after successful execution.

14 Appendices A–C

Appendix A: Virtual Machine Setup

On Ubuntu 18.04 "Bionic Beaver", run the commands in Listing 24 to get a basic running Kubernetes cluster with the essential client-side tools to install packaged and unpackaged Kubernetes applications.

Note: Tutorial participants using the pre-provisioned VMs do not need to enter these commands.

Listing 24: Basic Setup

```
sudo apt-get upgrade -u
sudo snap install microk8s --classic
sudo snap install kubectl --classic
sudo snap install helm --classic
kubectl cluster-info dump # check ok
helm init
kubectl get pods --namespace kube-system # check ok with
    tiller being installed

curl -L https://github.com/kubernetes/kompose/releases/
    download/v1.18.0/kompose-linux-amd64 -o kompose
curl -L https://github.com/kedgeproject/kedge/releases/
    download/v0.12.0/kedge-linux-amd64 -o kedge
chmod +x kompose kedge && sudo mv kompose kedge /usr/local/
    bin/
```

Listings 25 and 26 reference further possible client-side enhancements of the system to get `kubectl` plugins and the stand-alone `kustomize` tool which has functionality beyond applying patches.

Listing 25: Kubectl Plugins

```
(
  set -x; cd "$(mktemp -d)" &&
  curl -fsSLO "https://storage.googleapis.com/krew/v0.2.1/
      krew.{tar.gz,yaml}" &&
  tar zxvf krew.tar.gz &&
  ./krew-"$(uname | tr '[:upper:]' '[:lower:]')_amd64"
      install \
    --manifest=krew.yaml --archive=krew.tar.gz
)

export PATH="${KREW_ROOT:-$HOME/.krew}/bin:$PATH"
kubectl krew update
kubectl krew search
kubectl krew install access-matrix
kubectl access-matrix
```

Listing 26: Other Setup

```
sudo snap install go --classic
go get sigs.k8s.io/kustomize
# Kustomize installation might not work
```

For data anlytics, more packages need to be installed as shown in Listing 27.

Listing 27: Analysis Setup

```
sudo apt-get install python3-pandas
```

Appendix B: Sample Application

Listing 28 contains the YAML configuration for three application objects (namespace, deployment, service) which together form a user-facing web application.

Listing 28: Sample Application

```
apiVersion: v1
kind: Namespace
metadata:
  name: closer2020-whale-namespace
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: closer2020-whale-deployment
  namespace: closer2020-whale-namespace
spec:
  selector:
    matchLabels:
```

```
      dummy: redundant
  template:
    metadata:
      labels:
        dummy: redundant
    spec:
      containers:
      - image: crccheck/hello-world
        name: closer2020-whale-container
        ports:
        - containerPort: 8000
---
apiVersion: v1
kind: Service
metadata:
  name: closer2020-whale-service
  namespace: closer2020-whale-namespace
spec:
  ports:
  - nodePort: 30800
    port: 8000
    protocol: TCP
    targetPort: 8000
  type: NodePort
```

Appendix C: Dockerfile with Issues

Compact form (without comments) of the Dockerfile specifically crafted to contain many issues.

Listing 29: Dockerfile

```
FROM ubuntu:latest
FROM alpine:3.3
FROM anapsix/docker-alpine-java:8
RUN apk update
RUN apk add wget
RUN apk add git
RUN apk add --update wget git && rm -rf /var/cache/apk/*
RUN apk add --no-cache wget
RUN wget http://example.com/my/large/app.tar.gz -O /root/app
    .tar.gz
RUN mkdir /opt/somedir
RUN mv /root/app.tar.gz /opt/somedir/app.tar.gz
RUN tar -zxvf /opt/somedir/app.tar.gz
RUN mkdir /opt/somedir
RUN wget -q -O- http://example.com/my/large/app.tar.gz | tar
     -zxv -C /opt/somedir/
RUN workdir /opt/somedir
```

```
ADD http://example.com/my/large/app.tar.gz .
RUN ./bin/initapp.sh
ADD somefile.txt .
COPY super_secret_database.conf /opt/rails/config/database.
    yml
COPY entrypoint.sh .
ENTRYPOINT ["/entrypoint.sh"]
ENTRYPOINT ["/entrypoint.sh", "--db localhost", "--pass
    horriblepassword", "--user root"]
ENTRYPOINT ["/entrypoint.sh"]
CMD ["--db", "localhost", "--pass", "horriblepassword", "--
    user", "root"]
ENTRYPOINT nginx
RUN ln -s /dev/stdout /var/log/nginx/access.log && ln -s /
    dev/stderr /var/log/nginx/error.log
RUN nginx -g "daemon off"
RUN apk add --update nginx ruby
RUN /opt/rails/bin/rails s
ENTRYPOINT nginx -g "daemon off"
ENV ETCD_DATA_DIR /etcd
VOLUME $ETCD_DATA_DIR
EXPOSE 2379
```

References

[1] A. Daghighi. Introducing the Docker Compose Validator. ZHAW SPLab research blog: `https://blog.zhaw.ch/splab/2019/10/04/introducing-the-docker-compose-validator/`, October 2019.

[2] P. Gkikopoulos. Presenting the MAO Orchestrator. ZHAW SPLab research blog: `https://blog.zhaw.ch/splab/2019/10/22/presenting-the-mao-orchestrator/`, October 2019.

[3] V. S. Panagiotis Gkikopoulos, Josef Spillner. Data Distribution and Exploitation in a Global-Scale Microservice Artefact Observatory. EuroSys DW 2020, April 2020.

[4] I. A. Qasse, J. Spillner, M. A. Talib, and Q. Nasir. A Study on DApps Characteristics. IEEE DAPPS, August 2020.

[5] J. Spillner. Docker image checks: Quality, security, up-to-dateness, layers and inheritance. ZHAW SPLab research blog: `https://blog.zhaw.ch/splab/2019/11/15/docker-image-checks-quality-security-up-to-dateness-layers-and-inheritance/`, November 2019.

[6] J. Spillner. Quality Assessment and Improvement of Helm Charts for Kubernetes-Based Cloud Applications. arχiv:1901.00644, January 2019.

[7] J. Spillner. Quantitative Analysis of Cloud Function Evolution in the AWS Serverless Application Repository. arχiv:1905.04800, May 2019.