David Curran

1 Follower · About Follow

Microservices Running in Kubernetes Powered by Jenkins - Part 1

D David Curran Aug 30, 2018 · 7 min read

Working in a DevOps world subjects you to all kind of buzz words and technologies that have different meanings depending on the context they're in, such as those in my title.

With this in mind I'll start by giving definitions that will be used in context of this article, hopefully they should at least roughly match your own idea of what they are.

Microservice — A small program that delivers a service. This could be run as a container, a python script as part of a larger program, a binary running on a physical server etc. They will normally provide their service through a web accessible API. In this article we're using containers deployed in Kubernetes (K8s).

Kubernetes — An open-source system for automating deployment, scaling, and management of containerized applications. kubernetes.io. For the purposes of this article it can be thought of as a "container host" but what is shown should be scalable to make greater use of the K8s feature set.

Jenkins — Automation server. Again this article won't be utilising more than a small percentage of what Jenkins can do but should showcase how powerfull it can be. I'll be using it as a build and deployment server.

So what is the purpose of this article?

I've dabbled with Jenkins and Docker previously but without any kind of project to really get my teeth into with it I was unable to do anything particularly interesting. With some renewed vigor I came up with a small and simple project that could allow me to make use of Docker (in the form of K8s) and Jenkins as well as a NoSQL database (MongoDB).

Project and scope of the article

The project is, as I said, quite simple. There will be a container that creates some test score data and inputs this to MongoDB, another container will query this DB and return it as JSON via a REST API and another web container will display the information in a user friendly way.

I chose MongoDB as I have far less experience with non-relational databases than I do with relational ones such as MySQL so it provides some further learning. Also NoSQL databases are another "DevOps buzzword"

I won't cover how to install or configure any of these tools and make no promises about security best practices.

The data

I chose Python for the code base as it's a language I have plenty of experience in and find it great for knocking something together quickly. In terms of code structure I've kept all the python code and Dockerfiles for this project in one git repo for the sake of simplicity whereas these would normally be separate repos.

The data sent to MongoDB will look like:

```
[{
 'firstname': David,
 'surname': Curran,
 'age': 27,
 'score': 100
}]
```

To generate the data I created two lists, one with first names and one with surnames. I then pick at random from these lists and generate a random number between 18–65 for age and 0–100 for score.

```
from random import randint
class CreateData:
 …
def create(self):
 self.data['data'] = {}
 for x in range(0, self.points):
 self.data['data'][x] = {'firstname': self.fnames[randint(0,
(len(self.fnames)-1))],
 'surname': self.snames[randint(0, (len(self.snames)-1))],
 'age': randint(18, 65),
 'score': randint(0, 100)}
 return self.dat
```

Very simple, this will return a dict containing "self.points" number of records where self.points is any integer >0

This data is then put into MongoDB using the pymongo library

```
class ProcessData:
…
def process(self):
```

```
 response = {}
 response['result'] = {}
 num = 0
 for i in self.data['data']:
 num += 1
 result = self.db[self.database].insert_one(self.data['data'][i])
 response['result'][num] = {'id': result.inserted_id}
 response['count'] = num
 return response
```

Again, simple and could be improved upon no doubt but it does exactly what is required. It loops over the data from CreateData.create() and uses insert_one() to add it to MongoDB. To connect to MongoDB I've stored the user details in environment variables

```
client = MongoClient(os.getenv('MONGO_CLIENT'),
int(os.getenv('MONGO_PORT')))
db = client[self.database]
db.authenticate(os.getenv('MONGO_USER'), os.getenv('MONGO_PASS'))
```

The security flaw is quite obvious here and should not be used in a production environment but for the purposes of building a quick app it'll do.

Mircoservice

A the moment this isn't a microservice as per my definition up top. To make this so I created a Dockerfile

```
FROM alpine:3.7
MAINTAINER Schizoid90 "https://hub.docker.com/r/schizoid90"

COPY srcfiles/ /usr/local/src/

ENV LIBRARY_PATH=/lib:/usr/lib

RUN apk add - no-cache \
 python3 \
 && pip3 install - upgrade pip \
 && pip3 install -r /usr/local/src/processcode/requirements.txt

WORKDIR /usr/local/src

ENTRYPOINT [ "/usr/bin/python3", "-m", "main" ]
```

I'm using Alpine Linux to keep the resulting container image small (60.1MB). What this does exactly should be simple to work out. It installs python and uses pip to install the requirements from requirements.txt. It also uploads my application to /usr/local/src and runs this when the container starts. It calls the main.py file which is used as a wrapper and can take an argument "test" to run the tests (run by Jenkins). Without test it just runs the process normally.

Jenkins work

Jenkins is used to run the tests for this app, build the Docker image and upload it to my account on Docker Hub. At the moment this process is kicked off manually but writing a web hook that runs it on a commit to master is fairly trivial and not covered by this article.

Configuring the git repo

Here you see Jenkins is instructed to clone the dataproc git repo into the working directory.

I've used a plugin called "Environment Injector Plugin" in order to define the environment variables required by the Python script with the password variable encrypted automatically.

Set environment variables

Finally I've used execute shell to run the commands needed to build the image and push it to Docker Hub.

Test and build

This first runs the tests to make sure the code works and then builds and pushes the image. I had to create a sudo file for the Jenkins user to allow it to run the docker commands. The tests are, as ever, simple just to show the process. I also ran through the "docker login" process manually to make this work.

```
import os
from processcode.dataprocess import ProcessData
from .datacreate import CreateData
class RunTests:
…
def testing(self):
 cd = CreateData(self.points)
 data = cd.create()
```

```
 self.tests['datatotal'] = "PASS" if len(data['data']) == self.points
else "FAIL"
 pd = ProcessData(data = data)
 mongoinsert = pd.process()
 self.tests['mongoinsert'] = "PASS" if len(mongoinsert['result']) ==
self.points else "FAIL"
 self.tests['mongocount'] = "PASS" if mongoinsert['count'] ==
self.points else "FAIL"

…# code that checks if self.tests has any "FAIL" in.
```

The tests basically just make sure that the number of records in MongoDB matches the amount of data that was created.

Now we have a Docker image we can use with K8s we can use Jenkins to kick off a Kubernetes pod with our image to push data into MongoDB

I used the same git repo, variables and injection plugin as for the test and build so won't show that again. The real work happens in the "deploy to Kubernetes" build step.

Kubeconfig and build YAML

This utilises a Kubeconfig that directs Jenkins to the correct API endpoint for my Kubernetes cluster and a YAML file that configures the desired state for the data processing pod.

```
apiVersion: v1
kind: Pod
metadata:
```

```
    name: dataproc
  spec:
    containers:
    - name: dataproc
      image: schizoid90/dataproc
      env:
      - name: MONGO_CLIENT
        value: ${MONGO_CLIENT}
      - name: MONGO_PORT
        value: ${MONGO_PORT}
      - name: MONGO_USER
        value: ${MONGO_USER}
      - name: MONGO_PASS
        value: ${MONGO_PASS}
    restartPolicy: Never
```

I set up a Pod for this as I don't require any of the extra features Deployments and Jobs use, I just need to spin up a container that will do its job and then disappear. The ${X} variables under env: will be replaced by Jenkins using the Environment variables I set up there (as long as the option remains ticked in the build step). Another obvious security flaw here is that the MONGO_PASS variable will be available to `docker inspect` and `kubectl describe` but for the purposes of this project it will suffice. Pulling secrets from outside sources as an example was a bit too much work in this instance.

When I click "build now" in Jenkins, the pod spins up and runs the code. This deposits 200 documents into MongoDB.

```
# before
> db.data.count()
0

#after
> db.data.count()
200
```

So far I've not mentioned how I built the MongoDB instance. I'll explain that quickly here. I used the standard Mongo image "3.4-jessie" and created a service to pass traffic into the server to the pod.

```
IP:                     10.109.15.127
External IPs:           10.0.0.52
Port:                   27017/TCP
TargetPort:             27017/TCP
```

It's then as simple as forwarding a port of my choice to 27017 on the worker server in my cloud environment. I also created a Persistent volume and related claim but I'll go over that in a later part.

So now I've got a container that creates and deposits some data into a MongoDB instance. So far my microservice architecture isn't very useful, in Part 2 I'll cover how I interact with and consume this data.

Docker Kubernetes Microservices Jenkins DevOps