1st Lab - Cloud Computing and Distributed Systems

Authors

Oulis Evangelos	<cs151051@uniwa.gr>

Initiation of 1st Exercise

First of all we have to initiate out cluster configuration file that are **Dockerfile**, **package.json**, **dockerignore** and the **server.js** . To begin with, Dockerfile is a txt file that includes the instructions that required to perform the final docker image, some of these instructions are MKDIR to make the directory to include the final docker application, COPY to copy all the files that constitutes the image into the application image's files, EXPOSE to open the ports to the application and create and the CMD to start the application when the image is deployed.

Dockerfile

```
#From the official Python runtime as a paretn Image
#FROM python:2.7-slim

#From the official Node.js runtime as a pattern Image
FROM node:10

#Set the working directory to /app
WORKDIR /app

#Copy the current directory contents into the container at /app
COPY . /app

# Set instructions on build.
ONBUILD ADD package.json /app/
ONBUILD RUN npm install
ONBUILD ADD . /app

RUN npm install

#Install any needed packages specified in requirements.txt
#RUN pip install --trusted-host pypi.python.org -r requirements.txt

#Make port 80 available to the world outside this container
EXPOSE 80

#Define enviroment variable
ENV NAME World

#Run app.py when the conatiner launches
CMD ["npm", "start"]
```

Package is a JSON file that includes a discription of our application, the dependencies and the boot script with the application will start.

Package.json

```
{
    "name": "test1",
    "version": "1.0.0",
    "description": "Node.js on Docker",
    "author": "Oulis Evangelos <cs151051@uniwa.gr",
    "main": "server.js",
    "script": {
        "start": "node server.js"
    },
    "dependencies": {
        "express": "^4.16.1"
    }
}
```

Dockerignore is a rule file that includes the files that we don't want to use while the image executes.

dockerignore

```
node_modules
npm-debug.log
```

Application code is the file of the application implemented on Node.js .

server.js

```
'use script';
const express = require('express');

const PORT = '8080';
const HOST = '0.0.0.0';

const app = express();

app.get("/", (req, res) => {
    var os = require('os');
    let date_ob = new Date();
    let date = ("0" + date_ob.getDate()).slice(-2);
    let month = ("0" + (date_ob.getMonth() + 1)).slice(-2);
    let year = date_ob.getFullYear();
    let hours = date_ob.getHours();
    let minutes = date_ob.getMinutes();
    let seconds = date_ob.getSeconds();

    res.send('Hello world ' + os.hostname() + ', Current Date: ' + date + '/' + month
+ '/' + year + ' Hour: ' + hours + ':' + minutes + '.\n');
});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

All these files are uploaded to a GitHub repository that follows this link: repo also the docker image is uploaded to a DockerHub repository that follows this link: docker.

Upload Files to GitHub

To upload files on GitHub we have to create a local folder and initialize a GitHub repository. Then following the below instructions we are able to upload our work on GitHub.

```
$ git add .
$ git commit -a -m "1.1"
$ git push origin
```

The source tree of files that costructs the Docker Image is shown as below:

```
.
├─── Dockerfile
├─── dockerignore
├─── package.json
└─── server.js
```

Create a Local Docker Image

To test our application we are able create a local image and run it. To create and build the local image we have to execute the below instruction: **docker build -t ./Dockerfile** . This instruction must be execute into the application folder to have a success.

To check that the application built we should run the above instrunction and check the response, **docker image ls**. As output we have:

```
REPOSITORY              TAG                 IMAGE ID            CREATED
SIZE
oulievancs/first_lab    latest              c4a7d3d9a3a8        23 hours ago
914MB
ex1_node                latest              d1bd15d0b8fb        26 hours ago
914MB
oulievancs/first_lab    ex1_node            7a8f7e093fe7        36 hours ago
914MB
node                    10                  aa6432763c11        2 weeks ago
911MB
```

We see that the **ex1_node** is here and this is means that the image has already built.

Run the Local Image

To run the local image we have to execute the below instruction:

```
$ docker run -p 4000:8080 ex1_node
```

At the application name we have to give as a parameter the name of a docker as placed above. Then we have to check if it works hitting on a browser the http://localhost:4000 and if it response, the application will be fine.

Upload the Docker Image

To upload the docker image to DockerHub we have to create a docker repository using the docker web application and execute the below instructions:

```
$ docker login --username=yourhubusername --email=youremail@company.com     # to login
$ docker build -t oulievancs/first-lab -f ./Dockerfile .            # build the docker
file
$ docker tag oulievancs/first-lab:latest oulievancs/first_lab:v1        # create a tag
image of our application
$ docker push oulievancs/first-lab:v1                   # to push the image
```

The docker image uploaded at the repository on DockerHub and is available at the following link

DockerHub repo.

Deploy and Run the Docker Image on Google Gloud

To deploy and run the image we have to create an account to Google Cloud Platform GCP, see here for more information.

To run the docker image we have to link the GitHub account to the Google Cloud Platform more, and select the repository that contains our application's files. Then the GCP build and create the docker image and run it by creating a service.

In the other hand, we can push a new image into Googe Container Registry, deploy the image and expose the deployment as a Service.

- I will explain the second option.

Push the Image Into Google Registry

To prepare some value to work better.

```
$ export HOSTNAME=gcr.io
$ export PROJECT_ID=bustling-opus-262223
$ export IMAGE=hello-app
$ export TAG=v1
```

Then we able to build and push the new image to Google Registry using the below variables.

```
$ docker build -t ${HOSTNAME}/${PROJECT_ID}/${IMAGE}:${TAG}
$ docker push ${HOSTNAME}/${PROJECT_ID}/${IMAGE}:${TAG}
```

After that we have push the image onto Google Registry that is a storage that we can use to upload our built images to use for deployments.

Deploy Application and Create a Kubernete Pod

In this section is deliberate to talk about kubectl that is a user interface to interact with kubernetes nodes. Using this interface we are able to deploy the image and expose it onto the internet.

```
$ kubectl create deployment hello-web
--image=${HOSTNAME}/${PROJECT_ID}/${IMAGE}:${TAG}
```

A small description of the new pod is shown below:

```
o-web-6d4dfdd75b-4p6pr
Name:           hello-web-6d4dfdd75b-4p6pr
Namespace:      default
Priority:       0
Node:           gke-hello-world-default-pool-e6518ee4-kbhh/10.128.0.14
Start Time:     Wed, 18 Mar 2020 17:06:31 +0200
Labels:         app=hello-web
                pod-template-hash=6d4dfdd75b
Annotations:    kubernetes.io/limit-ranger: LimitRanger plugin set: cpu request for
container hello-app
Status:         Running
IP:             10.28.0.12
IPs:            <none>
Controlled By:  ReplicaSet/hello-web-6d4dfdd75b
Containers:
  hello-app:
    Container ID:
docker://eb1525e775216783203293e0e3403f8d93ffa63b96a111d7bcf322658e962782
    Image:          gcr.io/bustling-opus-262223/hello-app:v1
    Image ID:       docker-pullable://gcr.io/bustling-opus-262223/hello-
app@sha256:f73d303bc6b9ba0eb7c83039bea38cb5c85ce0377c82741d8239d0b9e7aea372
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Wed, 18 Mar 2020 17:07:04 +0200
    Ready:          True
    Restart Count:  0
    Requests:
      cpu:        100m
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-gbxxv (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-gbxxv:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-gbxxv
    Optional:    false
QoS Class:       Burstable
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age   From
Message
  ----    ------     ----  ----
```

```
-------
  Normal  Scheduled  109s  default-scheduler
Successfully assigned default/hello-web-6d4dfdd75b-4p6pr to gke-hello-world-default-
pool-e6518ee4-kbhh
  Normal  Pulling    108s  kubelet, gke-hello-world-default-pool-e6518ee4-kbhh
Pulling image "gcr.io/bustling-opus-262223/hello-app:v1"
  Normal  Pulled     80s   kubelet, gke-hello-world-default-pool-e6518ee4-kbhh
Successfully pulled image "gcr.io/bustling-opus-262223/hello-app:v1"
  Normal  Created    76s   kubelet, gke-hello-world-default-pool-e6518ee4-kbhh
Created container hello-app
  Normal  Started    76s   kubelet, gke-hello-world-default-pool-e6518ee4-kbhh
Started container hello-app
```

Expose The Deployment as a Service

Expose the demployment as a service is something like bridging the deployment with the external world. So after that you have the ability to call an application using an IP address.

```
$ kubectl expose deployment hello-web --type=LoadBalancer --port 80 --target-port 8080
```

A small description of our Service is shown below:

```
Name:                     hello-web
Namespace:                default
Labels:                   app=hello-web
Annotations:              <none>
Selector:                 app=hello-web
Type:                     LoadBalancer
IP:                       10.31.251.145
LoadBalancer Ingress:     35.184.146.255
Port:                     <unset>  80/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  30056/TCP
Endpoints:                10.28.0.12:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason                Age   From                Message
  ----    ------                ----  ----                -------
  Normal  EnsuringLoadBalancer  22m   service-controller  Ensuring load balancer
  Normal  EnsuredLoadBalancer   22m   service-controller  Ensured load balancer


Name:              kubernetes
Namespace:         default
Labels:            component=apiserver
                   provider=kubernetes
Annotations:       <none>
Selector:          <none>
Type:              ClusterIP
IP:                10.31.240.1
Port:              https  443/TCP
TargetPort:        443/TCP
Endpoints:         34.70.42.100:443
Session Affinity:  None
Events:            <none>
```

As we see the external-IP that we can use to access the service is 35.184.146.255 .

After deploy

After I deployed the docker image and expose it as a Service, I can access this sercice by hitting the endpoint external-IP on 80 port via a browser.

The link that was created is http://35.184.146.255:80 , so you can hit it.

Conclusion

In this report we describe the way to construct a Docker Application Image and upload it on the DockerHub. Moreover we discribe the steps to upload also an image on the Google Registry. Finaly

we descibe the way to deploy a Docker Image on Cloud and expose it as a Kubernete Service that you have the ability to have a remote access of a applicaation that runs on a Kubernetes cluster.

©