6. Docker Containers

DGP Solutions

Orchestration

- Orchestration is about the providing a means to have services collaborate with each other. Instead of building a single monolithic service to cover all aspects of a business flow, aspects of the system are replaced into their own re-useable services that collectively interact to represent the desired work flow.
- Orchestration provides centralized management of a resource pool of services and controlling interactions of work flow between them
- Cloud orchestration is the use of a programming technology to manage the interconnections and interactions

What is Docker?

- Docker is an open platform for developing, shipping, and running applications
- Docker provides the ability to package and run an application in a loosely isolated environment called a container. Container isolation allows you to run many containers simultaneously on a single given host.
- A virtual machine (VM) is an emulation of a computer system that can provide the functionality needed to execute an entire operating system. A hypervisor uses native executions to share and manage hardware, allowing for multiple environments which are isolated from one another, yet exist on the same physical machine.
- Containers, however, provide a way to run isolated systems on a single server/host and Operating System.

Docker cont.

- The Docker Engine is a client-server application with;
  - A server which is a type of long-running program called a daemon process that provides a network through which services communicate with each other
  - A REST API that specifies interfaces that programs can use to talk to the daemon and instruct it on what to do.
  - A command line interface (CLI) client (the docker command).
- The Docker registry stores Docker Images that are read-only templates with instructions for creating one or many Docker containers. These containers are runnable instances of an image. Our images are our services that have been effectively deployed to Docker. So we can “spin up” many isolated “instances” of these services (containers) behind their own Docker ports for scaling purposes.
- The Docker client talks to the Docker daemon which listens for Docker API requests and “orchestrates” Docker containers that represent our services

Steps to run a Service in Docker

- We will take a simple service that has been tested through the url http://localhost:8088/directory/town/4 to return a json object representing a town.


```
{
    "id": 4,
    "phoneCode": 1392,
    "district": "Budleigh Salterton",
    "city": "Exmouth",
    "county": "Devon",
    "station": null
}
```


- Now we are ready to place it under the control of Docker. First of all we do a mvn clean followed by a mvn install to generate an executable jar containing the Spring dependencies required to run the service.

DockerFile configuration

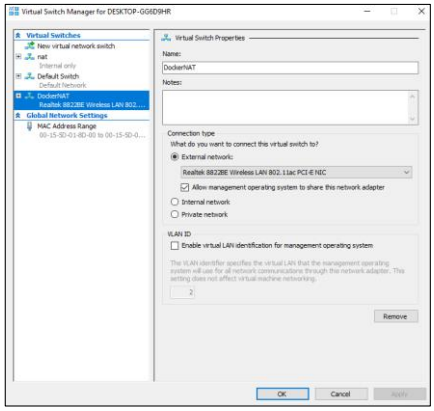
- Use a maven build with a goal of package to create a jar for your Spring boot application
- We now need a DockerFile as follows (Not the only way to do this);

```
FROM java:8
EXPOSE 8088
ADD /target/town-0.0.1-SNAPSHOT.jar town.jar
ENTRYPOINT  ["java", "-jar", "town.jar"]
```

- FROM – we need a jre as we are running Java so we get it form Docker java image
- EXPOSE – What is the port for our image (service)
- ADD – take the townl-0.0.1-SNAPSHOT jar and place it in the image as town.jar
- ENTRYPOINT – How do we get into the service. This is a Spring Boot application, it has a main method in a class in the jar

HyperV

- Install Docker for windows at https://docs.docker.com/docker-for-windows/install/. Once installed follow the web site instructions to start Docker
- We need some sort of Virtualization mechanism. With windows 10 that is **Hyper-V** . HyperV mist be enabled with a Virtual switch
- Open up the HyperV GUI and select “External Network” → Save
- RUN Powershell as Administrator. See if it recognizes the command docker-machine ls



Some simple Docker commands

- Stop containers
  - docker stop $(docker ps -aq) or $ docker rm $(docker ps --all -q)
- Remove containers
  - docker rm $(docker ps -aq)
- Remove an image
  - docker rmi employee
  - List images
- List images
  - docker images
- List running containers
  - docker ps

Image → Container

- After navigating to the projects directory where the DockerFile is, we build our image
  - $ docker build –f DockerFile –t town . (do not forget the dot)
  - $ docker images – this should show your new image called “town” in Docker’s registry
- Next, we want to create a runnable instance of the image or container. Since we are only dealing with one image we can use the run command.
- Note below I ask the container to expose the port 2222 which will delegate to port 8088 which is the port for our Entry Point from the DockerFile into our service
  - $ docker run -p 2222:8088 -d town
  - $ docker ps to confirm the container is running

Containers Running

- Then access our container service, via the url of the Docker host url and the DOCKER exposed port, http://localhost:2222/directory/town/4

- For scalability, I'd like two containers please
  - $ docker run -p 2233:8088 -d town
  - http://localhost:2233/directory/town/4

Orchestration and WorkFlow

- The Docker run command is fine for a single container start up, but what happens if we want to spin up multiple different containers from different images. This would get tiresome and could fail if services have dependencies on other services.

- Remember that services are in different ISOLATED containers so how could they find their dependency services?

Services call other services

- So our service wants to interact with another service. The REST endpoint looks like this;

```
@GetMapping(path="/{id}", produces= {MediaType.APPLICATION_JSON_VALUE})
public ResponseEntity<Town> get(@PathVariable("id") long id) {
    private String BASE_URL = "http://localhost:7777/";
  try {
         Town = dao.getTown(id);
         Station = new RestTemplate()
               .getForObject(BASE_URL + "{phone}" , Station.class, town.getPhoneCode());
         town.setStation(station);
         return ResponseEntity.ok(town);
  } catch (RuntimeException e) {
         System.out.println(e.getMessage());
        return ResponseEntity.notFound().build();
  }
}
```

19920612 Copyright DGP Solutions 6-12

- Dependency Service

```
@GetMapping(path="/{phone}", produces= {MediaType.APPLICATION_JSON_VALUE,
MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<Station> get(@PathVariable("phone") int id) {
 try {
 Station station = dao.get(id);
 return ResponseEntity.ok(station);
 } catch (RuntimeException e) {
 System.out.println(e.getMessage());
 return ResponseEntity.notFound().build();
 }
}
```

Another Image

- We will need a DockerFile and image created from the new service

```
FROM java:8
EXPOSE 7777
ADD /target/station-0.0.1-SNAPSHOT.jar station.jar
ENTRYPOINT  ["java", "-jar", "station.jar"]
```

- Then build an image
  - $ docker build –f DockerFile –t station .
- Stop and remove the current containers

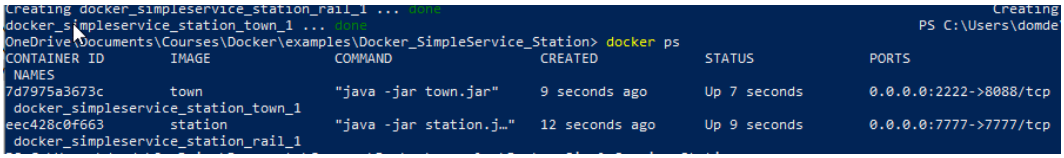
Define docker-compose

- To ensure that running containers of our two images can communicate with each other, we use the docker-compose command
- First make a docker-compose.yml file
- We have two services (call them anything)
- Each ties to a specific image
  - (remember our Docker build commands)
- Each with a Docker entry port
- And a port to the entry point of the image
- Then a “depends_on” entry
  - It uses the orchestration network of Docker’s daemon thread
  - To route a request from one container to another
  - In this case a container if another image

```
version: '2'
services:
  town:
    image: town
    ports:
     - "2222:8088"
    depends_on:
     - rail
  rail:
    image: station
    ports:
     - "7777:7777"
```


Execute docker-compose

- Now create the containers
  - $ docker-compose up –d
  - Creates containers for both images with one command and notifies the docker network of any dependencies



Discovering services

- By default docker-compose sets up a **single network** for your app.
- Each container for a service joins the **network** and is both reachable by other containers in that **network**
- It is discoverable by a **hostname identical to the container name** (i.e. SERVICE).
- This means we must change the url on the RestTemplate in the calling service to reflect the “service” name from docker-compose

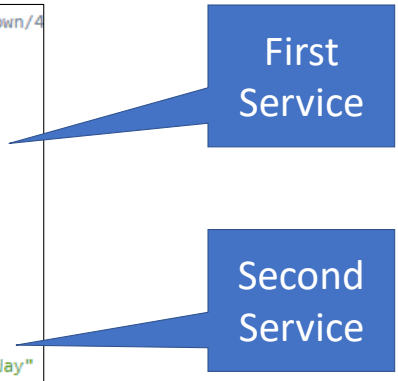
```
private String BASE_URL = http://rail:7777/
//rail is the name of the service in docker-compose
Station station = new RestTemplate().getForObject(BASE_URL + "{phone}" ,
Station.class, town.getPhoneCode());
```


- Now we will need to do a new maven build, docker-build

Service WorkFlow

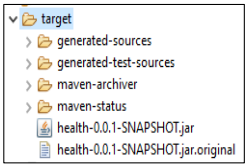
```
// http://localhost:2222/directory/town/4

{
  "id": 4,
  "phoneCode": 1392,
  "district": "Budleigh Salterton",
  "city": "Exmouth",
  "county": "Devon",
  "station": {
    "id": 1,
    "phoneCode": 1392,
    "name": "Exeter St Davids",
    "managedBy": "Great Western RailWay"
  }
}
```



Lab 6.1 Docker

Lab 6.1 – Docker Containers



- Objective: Deploy a Spring Boot Application to Docker
- Ensure docker desktop is running (See https://www.docker.com/products/docker-desktop for download instructions)
- First of all, we need jar. Select your project and build using maven with a goal of package. This will give you two jars, we want the "fat" jar with dependencies i.e. health-0.0.1-SNAPSHOT.jar

Lab 6.1 - Dockerfile

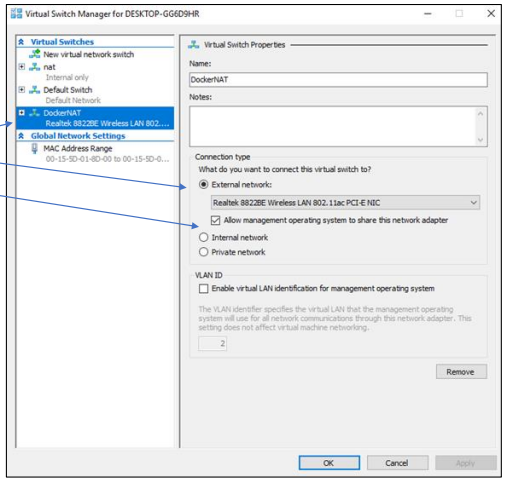
- Now create a file DockerFile directly under your Project node
- Place into the file the following

```
FROM java:8
EXPOSE 8082
ADD /target/health-0.0.1-SNAPSHOT.jar
health.jar
ENTRYPOINT  ["java", "-jar", "health.jar"]
```

- We are pulling the java image, exposing port 8082 as per your application.yaml
- Taking the jar and calling it something else
- The entry point is the java command which should launch you sPring boot Application

Lab 6.1 – Windows 10

- HyperV must be enabled with a Virtual switch, see; https://docs.docker.com/machine/drivers/hyper-v/#example
- Launch Docker Desktop – this may take **some time**



Lab 6.1 – Create Image

- In PowerShell navigate to your project directory (top node) check that docker is running
  - $ docker-machine ls

```
PS C:\Users\domde\OneDrive\Documents\eclipseWorkspaces\Webage_SB\Lab06.1_DockerContainer> docker-machine ls
NAME   ACTIVE   DRIVER   STATE   URL   SWARM   DOCKER   ERRORS
```

- Create an Image
  - $ docker build -f DockerFile -t healthplan . (DO NOT forget the dot!, ALL lowercase for the image name)
  - Check that the image is there with $ docker images

```
PS C:\Users\domde\OneDrive\Documents\eclipseWorkspaces\Webage_SB\Lab06.1_DockerContainer> docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
healthplan                          latest              f48ec9fb67d9        2 minutes ago       688MB
```


Housekeeping
Stop running containers
$ docker stop $(docker ps -aq) OR
$ docker rm $(docker ps --all -q)

Remove containers
docker rm $(docker ps -aq)

Remove an Image
$ docker rmi health
(You cannot remove an image that has running containers)

Lab 6.1 – Create a Container

- In PowerShell create a Container
  - $ docker run -p 8882:8082 -d healthplan
  - Note we expose port 8882 that delegates to the port 8082 of our container
  - Check that we have a running container status must be up
  - $ docker ps
  - Status must be up

```
PS C:\Users\domde\OneDrive\Documents\eclipseWorkspaces\Webage_SB\Lab06.1_DockerContainer> docker run -p 8882:8082 -d healthplan
cbf52072445d8f77c5c1cb65c3335c238a627fe15a8c57d7ae93d85ee71f5f28
PS C:\Users\domde\OneDrive\Documents\eclipseWorkspaces\Webage_SB\Lab06.1_DockerContainer> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
cbf52072445d        healthplan          "java -jar health.jar"   18 seconds ago      Up 11 seconds       0.0.0.0:8882->8082/tcp   pensive_khayyam
```


Lab 6.1 - Testing

- In a browser navigate to; http://localhost:8882/insure/healthPlan/1

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <plan>
    <copay>30.0</copay>
    <deductableFamily>4500</deductableFamily>
    <deductableIndividual>1500</deductableIndividual>
    <id>1</id>
    <name>LPPO</name>
    <outOfPocketFamily>10500</outOfPocketFamily>
    <outOfPocketIndividual>3500</outOfPocketIndividual>
    <zip>75024</zip>
  </plan>
```

- And, http://localhost:8882/insure/healthPlan


```
[{"id":3,"zip":75024,"name":"HMO","deductableIndividual":1000,"deductableFamily":3000,"outOfPocketIndividual":3000,"outOfPocketFamily":9000,"copay":10.0},
{"id":5,"zip":77429,"name":"HMO","deductableIndividual":3000,"deductableFamily":9000,"outOfPocketIndividual":6300,"outOfPocketFamily":12700,"copay":30.0},
{"id":7,"zip":78741,"name":"HMO","deductableIndividual":1500,"deductableFamily":4500,"outOfPocketIndividual":3500,"outOfPocketFamily":10500,"copay":
{"id":10,"zip":78666,"name":"HMO","deductableIndividual":3200,"deductableFamily":9750,"outOfPocketIndividual":3250,"outOfPocketFamily":9750,"copay"
{"id":15,"zip":79416,"name":"HMO","deductableIndividual":1500,"deductableFamily":4500,"outOfPocketIndividual":3500,"outOfPocketFamily":10500,"copay
{"id":18,"zip":79601,"name":"HMO","deductableIndividual":0,"deductableFamily":0,"outOfPocketIndividual":4800,"outOfPocketFamily":13200,"copay":30.0
{"id":19,"zip":79601,"name":"HMO","deductableIndividual":1000,"deductableFamily":3000,"outOfPocketIndividual":3000,"outOfPocketFamily":9000,"copay"
```


What would happen if you run the docker command?
$ docker run -p 8883:8082 -d healthplan
A second instance?