Distributed Systems
236351
Tutorial 6
Dockers

Yehonatan Buchnik
Technion
November 29, 2018
Overview

Docker is a platform for developing, shipping, and running applications.

- Enables to separate applications from infrastructure so one can deliver software quickly.
- Provides the ability to package and run an application in a loosely isolated environment called a **container**. The isolation and security allow to run many containers simultaneously on a given host.
**Docker Vs Virtual Machine**

Diagram:

- **CONTAINER**
  - App A
  - Bins/Libs
  - Docker
  - Host OS
  - Infrastructure

- **VM**
  - App A
  - Bins/Libs
  - Guest OS
  - Hypervisor
  - Infrastructure
## Docker Vs Virtual Machine

<table>
<thead>
<tr>
<th></th>
<th>VM’s</th>
<th>Docker containers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Based on</strong></td>
<td>Hypervisor</td>
<td>Docker engine (run directly within the host machines kernel)</td>
</tr>
<tr>
<td><strong>Isolation</strong></td>
<td>Strict</td>
<td>Loose</td>
</tr>
<tr>
<td><strong>Typical size</strong></td>
<td>GB’s</td>
<td>MB’s</td>
</tr>
<tr>
<td><strong>Startup time</strong></td>
<td>Minutes</td>
<td>Seconds</td>
</tr>
<tr>
<td><strong>Use cases</strong></td>
<td>Production of ”heavy systems” e.g, datacenter</td>
<td>Development, testing and distributed of an applications</td>
</tr>
</tbody>
</table>
Docker engine

- A server which is a type of long-running program called a daemon process.
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client
Architecture

- The Docker daemon listens for Docker API requests and manages the Docker entities.
- The Docker client is the primary way to interact with Docker. When using commands such as `docker run`, the client sends these commands to the daemon, which carries them out.
- A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. When using the `docker pull` or `docker run` commands, the required images are pulled from the daemon’s configured registry.
Images and Containers

- **Image** An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, one may build an image which is based on the ubuntu image, but installs the Apache web server and an application, as well as the configuration details needed to make the application run.

- **Container** A container is a runnable instance of an image. One can create, start, stop, move, or delete a container using the Docker API or CLI, as well as controlling the isolation level of a containers network, storage, or other underlying subsystems from other containers or the host machine. When a container is removed, any changes to its state that are not stored in persistent storage disappear.
Dockerfile

- In a container, access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system.
- So one needs to map ports to the outside world, and be specific about what files he wants to copy in to that environment.
- Dockerfile defines an image as well as what is exposed to the outside world, what should be copied to the image and so on.
Example
Dockerfile implementation

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

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

# 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 environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Full spec can be found [here](#)
Example

requirements.txt

We need to create `requirements.txt` and `app.py`:

```plaintext
Flask
Redis
```
We need to create `requirements.txt` and `app.py`:

```python
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)
app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"
    html = "<h3>Hello {name}!</h3> <b>Hostname:</b> {hostname}<br/> <b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```
Example

Build and run the app

Build the image:

```bash
docker build -t friendlyhello .
```

Check the image has actually created:

```bash
docker image ls
```

Run the app:

```bash
docker run -p 4000:80 friendlyhello
```

Although the application defines port 80, but this is happening only in the container mind. The exposed port is 4000 and hence you should see a message when browsing to `localhost:4000`.
Example

Stop the app

List containers:

docker container ls

Stop container

docker container stop $ContainerID$
Just a moment...

What does it have to do with Distributed Systems?

- Distributed systems are composed from a few different pieces that compute some task together.
  - For example, distributed storage system is consist of storage servers, frontend servers, metadata servers and so on.
  - What about creating a group of docker containers such that each container plays a different role in the system??
  - Think automation...
  - Cool ah?

We will now review **Docker Services, Docker Swarms** and **Docker Stacks**.
Services
Overview

- In a distributed application, different pieces of the app are called services.
- Services are really just containers in production. A service only runs one image, but it codifies the way that image runs, e.g., ports, replicas, capacities and so on.
- Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.
- Luckily its very easy to define, run, and scale services with the Docker platform – just write a `docker-compose.yml` file
Consider the *friendlyhello* image from before, write the following *docker-compose.yml* file (The complete spec can be found [here](#)):

```yaml
version: "3"
services:
    web:
        # replace friendlyhello with your name and image details
        image: friendlyhello
        deploy:
            replicas: 5
            resources:
                limits:
                    cpus: "0.1"
                    memory: 50M
            restart_policy:
                condition: on-failure
        ports:
            - "4000:80"
        networks:
            - webnet
networks:
    webnet:
```

Init a *swarm* (we will cover swarms in details later)

```
docker swarm init
```

Run the service (under the name *getstartedlab*)

```
docker stack deploy -c docker-compose.yml getstartedlab
```

List the services (in the output, look for the web service, prepended with the app name)

```
docker service ls
```

List the service containers (tasks)

```
docker service ps getstartedlab_web
```

Browse to *localhost:4000* and refresh it a couple of times. You should see how the container name is changing (in a round-robin fashion).
When rescaling the app (by changing the replicas’ number for example) run again

```
docker stack deploy -c docker-compose.yml getstartedlab
```

Take down the service

```
docker stack rm getstartedlab
```

Take down the swarm

```
docker swarm leave --force
```
Swarms

Overview

- A swarm is a group of machines that run Docker and are joined into a cluster.
- The Docker’s commands are the same as before but now they are executed on the cluster by the *swarm manager*.
- In the Docker’s terminology the swarm’s machines are referred as *nodes*.
Swarms
Manager and Workers

- Swarm managers are the only machines in a swarm that can execute commands, or authorize other machines to join the swarm as *workers*.
- Swarm managers can use several strategies to run containers.
  - Filling the least utilized with containers (The *emptiest node* method).
  - Ensures that each node gets exactly one instance of the specified container (The *global* method).
- One can instruct the swarm manager to use these strategies in *docker-compose.yml*. 
Swarms
Manager and Workers

Make the current node a swarm manager

docker swarm init

Make a node a worker (run with in the node environment with the credentials received while creating the manager)

docker swarm join

Leaving a swarm

docker swarm leave
Swarms
Building a cluster

To create a cluster, one first has to create nodes, which are actually lightweight VM’s.

Create a node

docker-machine create --driver virtualbox $node_name$

List nodes

docker-machine ls

Change environment to some node’s environment

eval $(docker-machine env $node_name$)

Unset the environment to the current node

eval $(docker-machine env -u)
Swarms

Example

Let’s deploy the service from before on a cluster consists of two nodes.

Create two nodes

```
docker-machine create --driver virtualbox node1
docker-machine create --driver virtualbox node2
```

Create *friendlyhello* image on each of the new nodes (no need if the image can be pulled from a remote repository)

```
eval $(docker-machine env node1)
docker build -t friendlyhello .
eval $(docker-machine env node2)
docker build -t friendlyhello .
```
Example

Create a swarm manager and two workers.

```bash
Docker swarm init
eval $(docker-machine env node1)
docker swarm join
eval $(docker-machine env node2)
docker swarm join
eval $(docker-machine env -u)
```

Deploy the service

```bash
docker stack deploy -c docker-compose.yml getstartedlab
```

Browse to `node_ip:4000` to see the replica running.
Stacks

Overview

With Stacks we simply implement the previous ideas on a stack of services. By configuring `docker-compose.yml` one can deploy a complete distributed system that run containers on swarm’s nodes. Let’s see an example...
Define in the `docker-compose.yml` two (for our convenience identical) services:

```yaml
version: "3"
services:
  web1:
    image: friendlyhello
    deploy:
      replicas: 3
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "4000:80"
  webnet
...
Define in the `docker-compose.yml` two (for our convenience identical):

```yaml
... 
web2:
  image: friendlyhello
  deploy:
    replicas: 3
    resources:
      limits:
        cpus: "0.1"
        memory: 50M
    restart_policy:
      condition: on-failure
  ports:
    - "5000:80"
  networks:
    - webnet
networks:
    webnet:
```

Note that as both services run in the same stack they have to expose two different ports.
Stacks

Example

Deploy the stack as before. Can you see the effects?
Conclusion

For more details see Docker documentation.