QA Consulting

Docker Compose Configuration

TRAINING MATERIALS - MODULE HANDOUT

Contacts

robert.crutchley@qa.com
team.qac.all.trainers@qa.com

www.consulting.qa.com

Contents

Overview **2**
Defining a Service 2

Container Names **2**
Default Container Names 2
Setting the Container Name 3

Service Dependencies **3**

Build **4**
Usage 4
Image 4
Build Context 4
Build Arguments 5

Ports **5**
Target Port 5
Published Port 5
Port Protocol 6

Volumes **6**
Volumes 6
Bind Mounts 7

Tasks **8**
Download and Configure the Project 8
Compose File 8
Service Definitions 9
Database Connectivity 9
Version Control 10
Ports 10
Set Version 10
The Compose file we have created relies on a host environment variable, which is the version of the application, we will set this now. 10
Start the Service 11
Stop and Remove the Services 11
Implement Another Project 11

Overview

You can consider a service as container effectively, however a service may be scaled by Docker Compose to be multiple containers.

When creating these configurations Docker Compose considers your application as a service and manages it as such. This is because you can have multiple instances of a front end web application for instance, if you wanted to update the application, it's nice to be able to update that application all at once, as opposed to each instance of it.

Service configurations accomplish things very similar to the ordinary Docker commands that you would be running such as providing arguments, setting environment variables, publishing ports etc.

Defining a Service

A service can be created as a property of services in the configuration file. A service must either have a valid image or build property for Compose to be able to run it.

```
version: '3.7'
services:
 nginx:
   image: nginx:latest
```

Container Names

Default Container Names

By default, Compose will create container name for you by using the parent directory name, the service name and the number of the container. The number of the container is affected depending on the amount you have scaled to for your service.

```
[PARENT_DIRECTORY]_[SERVICE_NAME]_[CONTAINER_NUMBER]
```

```
my_project_my_service_1
```

Setting the Container Name

You can manually set the container name with the container_name property. If your are planning on scaling your service then you shouldn't use this property. Only use this property on a service that you are sure you are going to only have one instance of, like a database or a load balancer. This is because Compose will try to create another container with that same name and fail.

```
version: '3.7'
services:
 nginx:
   image: nginx:latest
   container_name: nginx
 database:
   image: mysql:5.7
   container_name: mysql
```

Service Dependencies

Services can sometimes depend on others, like an application depending on database. Some applications might not even run unless the other service is running as well. We can accommodate for this by letting Compose know that a particular service depends on another with the depends_on property. Compose will wait until the service has started successfully before trying to start the dependant service.

```
version: '3.7'
services:
my_service:
  image: my_service:latest
  depends_on:
  - database
database:
  image: mysql:5.7
  container_name: mysql
```

Build

You are able to build your own images with Docker Compose by using the build property on a service. If the image can't be found in the local Docker registry then Compose will attempt to build it.

Usage

The most simple way to implement this property is by specifying the build context, which is where the Dockerfile is. The context location is relevant to the location of the docker-compose.yaml.

```
version: '3.7'
services:
 my_service:
   build: ./my_service
```

Image

If you have an image property defined, when the image is built, it will be named accordingly. If only the image property is provided the image will be pulled from the remote registries, skipping the build stage.

```
version: '3.7'
services:
 my_service:
   image: my_service:latest
   build: ./my_service
```

Build Context

The build context can be specified as a property of build if your are wanting to use the other build properties such as specifying build arguments for the Dockerfile.

```
version: '3'
services:
 my_service:
   build:
     context: ./my_service
```

Build Arguments

Just like with the regular Docker commands we can pass build arguments to the Dockerfile

```
version: '3.7'
services:
 my_service:
   build:
     context: ./my_service
     args:
       version: 1
```

Ports

Quite often we need to publish ports from the container to the host to be able to access the service. There are two different syntaxes for accomplishing this in a Compose configuration file, short and long syntaxes. We will be using the long syntax here as it is a very clear and intuitive way of publishing ports.

Target Port

A port mapping must at least have a target property, which is the port from inside the container which will be published. The target port will be published to a random high port number on the host if only this property is applied.

```
version: '3.7'
services:
 nginx:
   image: nginx:latest
   ports:
   - target: 80
```

Published Port

This property for the port mapping is the port on the host which the container port will be mapped to. Make sure that this port is not in use by another application or else it will not work.

```
version: '3.7'
services:
 nginx:
   image: nginx:latest
   ports:
   - target: 80
     published: 80
```

Port Protocol

The protocol is optional in most cases, by default it is TCP. We can set it however to be more verbose about the type of service that is listening. Whether your application is accepting traffic via UDP or TCP, setting this property will make it very clear.

```
version: '3.7'
services:
 nginx:
   image: nginx:latest
   ports:
   - target: 80
     published: 80
     protocol: tcp
```

Volumes

Volumes are excellent for persisting data and managing configurations across containers. Bind mounts can also be used for easily placing files and folders from the host into containers. Volumes must be defined under the volumes section in the Compose file.

Volumes

```
version: '3.7'
services:
 nginx:
   image: jenkins:latest
   volumes:
   - type: volume
     source: jenkins
     target: /var/jenkins_home
volumes:
 jenkins:
```

Bind Mounts

```
version: '3.7'
services:
 nginx:
   image: nginx:latest
   volumes:
   - type: bind
     source: ./nginx.conf
     target: /etc/nginx/nginx.conf
```

Tasks

This exercise will be using the OLS case study as a reference for resolving some of the challenges on a project using Docker Compose service configurations.

Download and Configure the Project

Before we can get started we need to download the OLS project, we can access it from the public GitHub repository (https://github.com/original-lizard-studios/website).

```
cd ~; git clone https://github.com/original-lizard-studios/website; cd website
```

At the root of the project we need to make a Dockerfile.

~/website/Dockerfile

```
FROM node:10 as client-build
WORKDIR /build
COPY client .
RUN npm install
RUN npm run build
FROM maven as server-build
WORKDIR /build
COPY server .
COPY --from=client-build /build/build src/main/resources/static
RUN mvn clean package
FROM java:8
WORKDIR /opt/app
COPY --from=server-build /build/target/app-0.0.1-SNAPSHOT.jar app.jar
EXPOSE 8080
ENTRYPOINT ["/usr/bin/java", "-jar", "app.jar"]
```

Compose File

It's important to set the version of your Compose configuration, otherwise some features might not work. Create the docker-compose.yaml file at the root of the project and set the version to 3.7.

~/website/docker-compose.yaml

```
version: '3.7'
```

Service Definitions

We are going to have two services deployed, the application itself and the Mongo Database. We need to create a build context to the application, seen as we are creating the images ourselves and set an image for the Mongo Database to be downloaded.

~/website/docker-compose.yaml

```
version: '3.7'
services:
 app:
   build: .
 mongo:
   image: mongo:latest
```

Database Connectivity

There is only going to be one instance of the database, so we can configure the container name for the Mongo service. We would do this because Compose is going to have both the services on the same Docker network. The application will be able to connect to the Mongo database using the DNS, the containers name will resolve to the private Docker network IP.

~/website/docker-compose.yaml

```
version: '3.7'
services:
app:
  image: app:${APP_VERSION}
  build: .
mongo:
  image: mongo:latest
  container_name: mongo
```

Version Control

OLS need to be able to update and rollback versions of the application smoothly. To do this we can use host environment variables to set a tag for the image property on the application service. Now when we are deploying, the version that is stored in APP_VERSION will be built and deployed.

~/website/docker-compose.yaml

```
version: '3.7'
services:
 app:
   image: app:${APP_VERSION}
   build: .
 mongo:
   image: mongo:latest
   container_name: mongo
```

Ports

To be able to access the application from the host we must publish the application port.

~/website/docker-compose.yaml

```
version: '3.7'
services:
 app:
   image: app:${APP_VERSION}
   build: .
   ports:
   - target: 8080
     protocol: tcp
 mongo:
   image: mongo:latest
   container_name: mongo
```

Set Version

The Compose file we have created relies on a host environment variable, which is the version of the application, we will set this now.

```
export APP_VERSION=v1
```

Start the Service

The service can be brought up with the up command in Compose.

```
docker-compose up -d
```

Stop and Remove the Services

Once you have viewed the application with a browser or a CLI tool like curl, stop and remove everything.

```
docker-compose down --rmi all
```

Implement Another Project

Try to have this run for your own project or one provided by the instructor. If the application is only a single Docker container, then you could try setting it up behind NGINX.