Dockerizing

Raúl Estrada

Octubre 2020

1. Overview

In this article, we'll focus on how to dockerize a *Spring Boot Application* to run it in an isolated environment, a.k.a. *container*.

Furthermore, we'll show how to create a composition of containers, which depend on each other and are linked against each other in a virtual private network. We'll also see how they can be managed together with single commands.

Let's start by creating a Java-enabled, lightweight base image, running *Alpine Linux*.

2. Buildpacks Support in Spring Boot 2.3

Spring Boot 2.3 added support for buildpacks. Put simply, instead of creating our own Dockerfile and building it using something like *docker build*, all we have to is to issue the following command:

```
$ ./mvnw spring-boot:build-image
```

Or in Gradle:

```
$ ./gradlew bootBuildImage
```

The main motivation behind buildpacks is to create the same deployment experience that some well-known cloud services such as Heroku or Cloud Foundry are providing for a while. We just run the *build-image* goal and the platform itself takes care of building and deploying the artifact.

Moreover, it can help us to change the way we're building Docker images more effectively. Instead of applying the same change to lots of Dockerfiles in different projects, all we have to do is to change or tune the buildpacks' image builder.

In addition to ease of use and better overall developer experience, it can be more efficient, too. For instance, the buildpacks approach will create a layered Docker image and uses the exploded version of the Jar file.

3. Common Base Image

We're going to be using *Docker's* own build-file format: a *Dockerfile*.

A *Dockerfile* is in principle, a linewise batch file, containing commands to build an image. It's not absolutely necessary to put these commands into a file, because we're able to pass them to the command-line, as well – a file is simply more convenient.

So, let's write our first *Dockerfile*:

```
FROM alpine:edge
MAINTAINER baeldung.com
RUN apk add --no-cache openjdk8
COPY files/UnlimitedJCEPolicyJDK8/* \
  /usr/lib/jvm/java-1.8-openjdk/jre/lib/security/
```

- **FROM**: The keyword *FROM*, tells *Docker* to use a given image with its tag as build-base. If this image is not in the local library, an online-search on ***DockerHub***, or on any other configured remote-registry, is performed
- **MAINTAINER**: A *MAINTAINER* is usually an email address, identifying the author of an image
- **RUN**: With the *RUN* command, we're executing a shell command-line within the target system. Here we utilizing *Alpine Linux's* package manager *apk* to install the *Java 8 OpenJDK*
- **COPY**: The last command tells *Docker* to *COPY* a few files from the local file-system, specifically a subfolder to the build directory, into the image in a given path

REQUIREMENTS: In order to run the tutorial successfully, you have to download the *Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files* from *Oracle*. Simply extract the downloaded archive into a local folder named *'files'*.

To finally build the image and store it in the local library, we have to run:

```
docker build --tag=alpine-java:base --rm=true .
```

NOTICE: The *–tag* option will give the image its name and *–rm=true* will remove intermediate images after it has been built successfully. The last character in this shell command is a dot, acting as a build-directory argument.

4. Dockerize a Standalone Spring Boot Application

As an example for an application which we can dockerize, we will take the *spring-cloud-config/server* from the spring cloud configuration tutorial. As a preparation-step, we have to assemble a runnable jar file and copy it to our *Docker* build-directory:

```
tutorials $> cd spring-cloud-config/server
server    $> mvn package spring-boot:repackage
server    $> cp target/server-0.0.1-SNAPSHOT.jar \
                 ../../spring-boot-docker/files/config-server.jar
server    $> cd ../../spring-boot-docker
```

Now we will create a *Dockerfile* named *Dockerfile.server* with the following content:

```
FROM alpine-java:base
MAINTAINER baeldung.com
COPY files/spring-cloud-config-server.jar /opt/spring-cloud/lib/
COPY files/spring-cloud-config-server-entrypoint.sh /opt/spring-cloud/bin/
ENV SPRING_APPLICATION_JSON= \
  '{"spring": {"cloud": {"config": {"server": \
  {"git": {"uri": "/var/lib/spring-cloud/config-repo", \
  "clone-on-start": true}}}}}}'
ENTRYPOINT ["/usr/bin/java"]
CMD ["-jar", "/opt/spring-cloud/lib/spring-cloud-config-server.jar"]
VOLUME /var/lib/spring-cloud/config-repo
EXPOSE 8888
```

- **FROM**: As base for our image we will take the *Java*-enabled *Alpine Linux*, created in the previous section
- **COPY**: We let *Docker* copy our jar file into the image
- **ENV**: This command lets us define some environment variables, which will be respected by the application running in the container. Here we define a customized ***Spring Boot Application* configuration**, to hand-over to the jar-executable later
- **ENTRYPOINT/CMD**: This will be the executable to start when the container is booting. We must define them as *JSON-Array*, because we will use an *ENTRYPOINT* in combination with a *CMD* for some application arguments
- **VOLUME**: Because our container will be running in an isolated environment, with no direct network access, we have to define a mountpoint-placeholder for our configuration repository
- **EXPOSE**: Here we are telling *Docker*, on which port our application is listing. This port will be published to the host, when the container is booting

To create an image from our *Dockerfile*, we have to run *'docker build'*, like before:

```
$> docker build --file=Dockerfile.server \
     --tag=config-server:latest --rm=true .
```

But before we're going to run a container from our image, we have to create a volume for mounting:

```
$> docker volume create --name=spring-cloud-config-repo
```

NOTICE: While a container is immutable, when not committed to an image after application exits, data stored in a volume will be persistent over several containers.

Finally, we are able to run the container from our image:

```
$> docker run --name=config-server --publish=8888:8888 \
     --volume=spring-cloud-config-repo:/var/lib/spring-cloud/config-repo \
     config-server:latest
```

- First, we have to ***–name*** our container. If not, one will be automatically chosen
- Then, we must ***–publish*** our exposed port (see *Dockerfile*) to a port on our host. The value is given in the form *'host-port:container-port'*. If only a container-port is given, a randomly chosen host-port will be used. If we leave this option out, the container will be completely isolated
- The ***–volume*** option gives access to either a directory on the host (when used with an absolute path) or a previously created *Docker* volume (when used with a *volume-name*). The path after the colon specifies the *mountpoint* within the container
- As argument we have to tell *Docker*, which image to be used. Here we have to give the *image-name* from the previously '*docker build*' step
- Some more useful options:
  - **-it** – enable interactive mode and allocate a *pseudo-tty*
  - **-d** – detach from the container after booting

If we run the container in detached mode, we can inspect its details, stop it and remove it with the following commands:

```
$> docker inspect config-server
$> docker stop config-server
$> docker rm config-server
```

5. Dockerize Dependent Applications in a Composite

Docker commands and *Dockerfiles* are particularly suitable for creating individual containers. But if you want to *operate on a network of isolated applications*, the container management quickly becomes cluttered.

To solve that, *Docker* provides a tool named *Docker Compose*. This comes with an own build-file in *YAML* format and is better suited in managing multiple containers. For example, it is able to start or stop a composite of services in one command, or merges the logging output of multiple services together into one *pseudo-tty*.

Let's build an example of two applications running in different Docker containers. They will communicate with each other and be presented as a "single unit" to the host system. We will build and copy the *spring-cloud-config/client* example described in the spring cloud configuration tutorial to our *files* folder, like we have done before with the *config-server*.

This will be our *docker-compose.yml*:

```
version: '2'
services:
    config-server:
        container_name: config-server
        build:
            context: .
            dockerfile: Dockerfile.server
        image: config-server:latest
        expose:
            - 8888
        networks:
            - spring-cloud-network
        volumes:
            - spring-cloud-config-repo:/var/lib/spring-cloud/config-repo
        logging:
            driver: json-file
```

- **version**: Specifies which format version should be used. This is a mandatory field. Here we use the newer version, whereas the *legacy format* is '1'
- **services**: Each object in this key defines a *service*, a.k.a container. This section is mandatory
  - **build**: If given, *docker-compose* is able to build an image from a *Dockerfile*
    - **context**: If given, it specifies the build-directory, where the *Dockerfile* is looked-up
    - **dockerfile**: If given, it sets an alternate name for a *Dockerfile*
  - **image**: Tells *Docker* which name it should give to the image when build-features are used. Otherwise, it is searching for this image in the library or *remote-registry*
  - **networks**: This is the identifier of the named networks to use. A given *name-value* must be listed in the *networks* section
  - **volumes**: This identifies the named volumes to use and the mountpoints to mount the volumes to, separated by a colon. Likewise in *networks* section, a *volume-name* must be defined in separate *volumes* section

```
    config-client:
        container_name: config-client
        build:
            context: .
            dockerfile: Dockerfile.client
        image: config-client:latest
        entrypoint: /opt/spring-cloud/bin/config-client-entrypoint.sh
        environment:
            SPRING_APPLICATION_JSON: \
              '{"spring": {"cloud":  \
              {"config": {"uri": "http://config-server:8888"}}}}'
        expose:
            - 8080
        ports:
            - 8080:8080
        networks:
            - spring-cloud-network
        links:
            - config-server:config-server
        depends_on:
            - config-server
        logging:
            driver: json-file
networks:
    spring-cloud-network:
        driver: bridge
volumes:
    spring-cloud-config-repo:
        external: true
```

    - **links**: This will create an internal network link between *this* service and the listed service. *This* service will be able to connect to the listed service, whereby the part before the colon specifies a *service-name* from the *services* section and the part after the colon specifies the hostname at which the service is listening on an exposed port
    - **depends_on**: This tells *Docker* to start a service only, if the listed services have started successfully. **NOTICE:** This works only at container level! For a workaround to start the dependent *application* first, see *config-client-entrypoint.sh*
    - **logging**: Here we are using the *'json-file'* driver, which is the default one. Alternatively *'syslog'* with a given address option or *'none'* can be used
- **networks**: In this section we're specifying the *networks* available to our services. In this example, we let *docker-compose* create a named *network* of type *'bridge'* for us. If the option *external* is set to *true*, it will use an existing one with the given name
- **volumes**: This is very similar to the *networks* section

Before we continue, we will check our build-file for syntax-errors:

```
$> docker-compose config
```

This will be our *Dockerfile.client* to build the *config-client* image from. It differs from the *Dockerfile.server* in that we additionally install *OpenBSD netcat* (which is needed in the next step) and make the *entrypoint* executable:

```
FROM alpine-java:base
MAINTAINER baeldung.com
RUN apk --no-cache add netcat-openbsd
COPY files/config-client.jar /opt/spring-cloud/lib/
COPY files/config-client-entrypoint.sh /opt/spring-cloud/bin/
RUN chmod 755 /opt/spring-cloud/bin/config-client-entrypoint.sh
```

And this will be the customized *entrypoint* for our *config-client service*. Here we use *netcat* in a loop to check whether our *config-server* is ready. You have to notice, that we can reach our *config-server* by its *link-name*, instead of an IP address:

```
#!/bin/sh
while ! nc -z config-server 8888 ; do
    echo "Waiting for upcoming Config Server"
    sleep 2
done
java -jar /opt/spring-cloud/lib/config-client.jar
```

Finally, we can build our images, create the defined containers, and start it in one command:

```
$> docker-compose up --build
```

To stop the containers, remove it from *Docker* and remove the connected *networks* and *volumes* from it, we can use the opposite command:

```
$> docker-compose down
```

A nice feature of *docker-compose* is the **ability to scale services**. For example, we can tell *Docker* to run one container for the *config-server* and three containers for the *config-client*.

But for this to work properly, we have to remove the *container_name* from our *docker-compose.yml*, for letting *Docker* choose one, and we have to change the *exposed port configuration*, to avoid clashes.

After that, we are able to scale our services like so:

```
$> docker-compose build
$> docker-compose up -d
$> docker-compose scale config-server=1 config-client=3
```

6. Conclusion

As we've seen, we are now able to build custom *Docker* images, running a *Spring Boot Application* as a *Docker* container, and creating dependent containers with *docker-compose*.

For further reading about the build-files, we refer to the official *Dockerfile reference* and the *docker-compose.yml reference*.