Docker
Aspectos adicionales

Raúl Estrada

Octubre 2020

Sharing directories

All state, including the filesystem, lives only for the lifetime of the container. When you *rm* the container, you destroy the state also.

If you want to preserve data, it needs to be stored outside of the container. Let’s demonstrate this by mapping the application log to a directory on the host system.

First, add a logback configuration to the application:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>/var/log/Application/application.log</file>
        <append>true</append>
        <encoder>
            <pattern>%-7d{yyyy-MM-dd HH:mm:ss:SSS} %m%n</pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

Then modify your Dockerfile to use it:

```
FROM openjdk:8-jre-alpine

COPY spring-boot-app-0.0.1-SNAPSHOT.war /app.war

COPY logback.xml /logback.xml

CMD ["/usr/bin/java", "-jar", "-Dspring.profiles.active=default",
  "-Dlogging.config=/logback.xml", "/app.war"]
```

You're copying the *logback.xml* into the image, and adding the logging configuration to the command line.

The logging configuration places the application logs in */var/log/Application/*.

Rebuild the image:

```
$ docker build -t spring-boot-app:latest .
Sending build context to Docker daemon  131.1MB
Step 1/4 : FROM openjdk:8-jre-alpine
 ---> c529fb7782f9
Step 2/4 : COPY target/spring-boot-app-0.0.1-SNAPSHOT.war /app.war
 ---> Using cache
 ---> d19bfa9fdfa7
Step 3/4 : COPY src/main/resources/logback.xml /logback.xml
 ---> Using cache
 ---> d62f97d9900d
Step 4/4 : CMD ["/usr/bin/java", "-jar", "-Dspring.profiles.active=default",
  "-Dlogging.config=/logback.xml", "/app.war"]
 ---> Using cache
 ---> fb9139a8c8b8
Successfully built fb9139a8c8b8
Successfully tagged spring-boot-app:latest
```

Docker didn't download the *openjdk:8-jre-alpine* image since *docker* has it cached locally.

Look at the build command. You specify an image tag with *-t*. This is the same tag that you pass to *docker run*. You provide the working directory last.

Now, you need to map the directory to a directory on the host when you run our container:

```
$ docker run -d -v /var/log/app:/var/log/Application/
  -p 8080:8080 spring-boot-app:latest
```

The *-v* option maps */var/log/app* on our host system to */var/log/Application/* in the container.

When you run this command, you can see a log file created in the mapped directory.

Naming Docker containers

You've been letting *docker* assign names to your containers. You can override this with *–name:*

```
$ docker run -d --name bootapp -v /var/log/app:/var/log/Application/
  -p 8080:8080 spring-boot-app:latest
57eb3f1998f503dc146d1f3b7ab9a6b221a939537be17ffc40fd410e2b72eda3
$ docker ps
IMAGE                    STATUS         PORTS                      NAMES
spring-boot-app:latest   Up 2 seconds   0.0.0.0:8080->8080/tcp     bootapp
```

Adding packages

When you looked at your image's history, you saw the command for adding the jre to Alpine. You can add packages to Alpine in your *Dockerfile*, too. Let's add *bash* to the container.

First, add the *APK* command to our *Dockerfile:*

```dockerfile
# Alpine Linux with OpenJDK JRE
FROM openjdk:8-jre-alpine
RUN apk add --no-cache bash

# Copy WAR
COPY spring-boot-app-0.0.1-SNAPSHOT.war /app.war

# copy fat WAR
COPY logback.xml /logback.xml

# runs application
CMD ["/usr/bin/java", "-jar", "-Dspring.profiles.active=default",
  "-Dlogging.config=/logback.xml", "/app.war"]
```

Then build the image with the same directives as before:

```
$ docker build -t spring-boot-app:latest .
Sending build context to Docker daemon     40MB
Step 1/5 : FROM openjdk:8-jre-alpine
 ---> c529fb7782f9
Step 2/5 : RUN apk add --no-cache bash
 ---> Using cache
 ---> 3b0c475c9bd0
Step 3/5 : COPY spring-boot-ops.war /app.war
 ---> c03bd6c6ace5
Step 4/5 : COPY logback.xml /logback.xml
 ---> b2f899ebec17
Step 5/5 : CMD ["/usr/bin/java", "-jar", "-Dspring.profiles.active=default",
  "-Dlogging.config=/logback.xml", "/app.war"]
 ---> Running in 3df30746d7a8
Removing intermediate container 3df30746d7a8
 ---> cbbfb596a092
Successfully built cbbfb596a092
Successfully tagged spring-boot-app:latest
```

The output is a little different this time. You can see where *bash* was installed in step two.

Finally, after you run the container, you can shell in with *bash:*

```
$ docker exec -it bootapp bash
bash-4.4# ls
app.war      etc          logback.xml  proc         sbin         tmp
bin          home         media        root         srv          usr
dev          lib          mnt          run          sys          var
bash-4.4#
```

Passing command line variables

So far, you've been running the Spring Boot application with the active profile set to default. You may want to build a single jar with different profiles and then select the correct one at runtime. Let's modify our image to accept the active profile as a command line argument.

First, create a shell script in the *docker* directory that runs the web application:

```
#!/bin/sh

java -Dspring.profiles.active=$1 -Dlogging.config=/logback.xml -jar /app.war
```

This script accepts a single argument and uses it as the name of the active profile.

Then, modify your *Dockerfile* to use this script to run the application:

```
# Alpine Linux with OpenJDK JRE
FROM openjdk:8-jre-alpine
RUN apk add --no-cache bash

# copy fat WAR
COPY spring-boot-app-1.0.0-SNAPSHOT.war /app.war

# copy fat WAR
COPY logback.xml /logback.xml

COPY run.sh /run.sh

ENTRYPOINT ["/run.sh"]
```

Dockerfile offers two mechanisms for starting a container; the *ENTRYPOINT* and the *CMD.* Simply put, the *ENTRYPOINT* is the program that is executed to start the container and *CMD* is the argument passed to that program.

The default *ENTRYPOINT is /bin/sh -c.* Until now, you were passing our Java command array to a shell.

Now, the dockerfile is copying the script to the image and then defining as the image's *ENTRYPOINT.* There is no *CMD.*

Build this image and then run it with *dev* as the final argument on the command line:

```
$ docker run -d --name bootapp -v /var/log/app:/var/log/Application/
  -p 8080:8080 spring-boot-app:latest dev
```

And then take a look at the logs for the active profile:

```
$ grep profiles /var/log/webapp/application.log
2018-06-11 00:33:50:016 The following profiles are active: dev
```

You can see that the profile setting was passed into the JVM.

Publishing images

We’ve only used the image on your development system. Eventually, you’ll want to distribute it to clients or production systems. This is done with a registry, where images are pushed with a name and tag and then pulled when they are run as containers. You saw this in action at the start of this tutorial when *docker* pulled the *hello-world* image for you.

The first step is to create an account on Docker Cloud. Go and create an account there if you don't already have one.

Next, log in to the Docker registry on our development system:

```
$ docker login
Username: baeldung
Password:
Login Succeeded
```

Next, tag the image. The format for tags is username/repository:tag. Tags and repository names are effectively freeform.

Tag the image and then list the images on your system to see the tag:

```
$ docker tag spring-boot-app baeldung/spring-boot-app:.0.0.1
$ docker image ls
REPOSITORY                              TAG                 IMAGE ID         CREATED
SIZE
spring-boot-app                         latest              f20d5002c78e     24 minutes ag
o       132MB
baeldung/spring-boot-app   1.00                    f20d5002c78e     24 minutes ago       1
32MB
openjdk                                 8-jre-alpine        c529fb7782f9     4 days ago
82MB
```

Note that the new image tag and the original image have the same image ID and size. Tags don't create new copies of images. They're pointers.

Now you can push the image to Docker Hub:

```
$ docker push baeldung/spring-boot-app:.0.0.1
The push refers to repository [docker.io/baeldung/spring-boot-app]
8bfb0f145ab3: Pushed
2e0170d39ba4: Pushed
789b0cedce1e: Pushed
f58f29c8ecaa: Pushed
cabb207275ad: Mounted from library/openjdk
a8cc3712c14a: Mounted from library/openjdk
cd7100a72410: Mounted from library/openjdk
1.00: digest: sha256:4c00fe46080f1e94d6de90717f1086f03cea06f7984cb8d6ea5dbc525e
3ecf27 size: 1784
```

Docker push accepts a tag name and pushes it to the default repository, which is Docker Hub.

Now, if you visit your account area on hub.docker.com, you can see the new repository, the image, and the tag.

Now you can pull the image down and run it on any system:

```
$ docker run -d --name bootapp -v /var/log/app:/var/log/Application/
  -p 8080:8080 ericgoebelbecker/spring-boot-app:.0.0.1 dev
Unable to find image 'baeldung/spring-boot-ops:1.00' locally
1.00: Pulling from baeldung/spring-boot-ops
b0568b191983: Pull complete
55a7da9473ae: Pull complete
422d2e7f1272: Pull complete
3292695f8261: Pull complete
Digest: sha256:4c00fe46080f1e94d6de90717f1086f03cea06f7984cb8d6ea5dbc525e3ecf27
Status: Downloaded newer image for baeldung/spring-boot-app:.0.0.1
```

This is the output of run on a different system from the one I built on. Similar to the way you ran hello-world, you passed the image tag to docker run. And since the image was not available locally, Docker pulled it from Docker Hub, assembled it, and ran it.

Conclusion

Docker is a robust platform for building, managing, and running containerized applications. In this tutorial, we installed the tools, packaged a Spring Boot application, looked at how we can manage containers and images, and then added some improvements to our application.

Finally, we published our image to Docker Hub, where it can be downloaded and run on any Docker-enabled host.