DZone

A Dockerfile for Maven-Based GitHub Projects

by Nicolas Frankel MVB · **Aug. 01, 17 · Java Zone**

Since the Docker "revolution," I've been interested in creating a `Dockefile` for Spring applications. I'm far from an ardent practitioner, but the principle of creating the `Dockerfile` is dead simple. As in Java, or probably any programming language, however, while it's easy to achieve something that works, it's much harder to create something that works well.

Multi-Stage Builds

In Docker, one of the main issues is the size of the final image. It's not uncommon to end up with images over 1 GB even for simple Java applications. Since version 17.05 of Docker, it's possible to have multiple builds in a single `Dockerfile`, and to access the output of the previous build into the current one. Those are called multi-stage builds. The final image will be based on the last build stage.

Let's imagine the code is hosted on GitHub, and that it's based on Maven. Build stages would be as follows:

1. Clone the code from GitHub.
2. Copy the folder from the previous stage; build the app with Maven.
3. Copy the JAR from the previous stage; run it with `java -jar`

Here is a build file to start from:

```
1  FROM alpine/git
2  WORKDIR /app
3  RUN git clone https://github.com/spring-projects/spring-petclinic.git (1)
4
5  FROM maven:3.5-jdk-8-alpine
6  WORKDIR /app
7  COPY --from=0 /app/spring-petclinic /app (2)
8  RUN mvn install (3)
9
10 FROM openjdk:8-jre-alpine
11 WORKDIR /app
12 COPY --from=1 /app/target/spring-petclinic-1.5.1.jar /app (4)
   CMD ["java -jar spring-petclinic-1.5.1.jar"] (5)
```

```
13 CMD [ "java -jar spring-petclinic-1.5.1.jar" ] (5)
```

It maps the above build stages:

1	Clone the Spring PetClinic git repository from Github
2	Copy the project folder from the previous build stage
3	Build the app
4	Copy the JAR from the previous build stage
5	Run the app

Improving Readability

Notice that in the previous build file, build stages are referenced via their index (starting from 0) *e.g.* `COPY --from=0` . While not a real issue, it's always better to have something semantically meaningful. Docker allows us to label stages — and references those labels in later stages.

```
1  FROM alpine/git as clone (1)
2  WORKDIR /app
3  RUN git clone https://github.com/spring-projects/spring-petclinic.git
4
5  FROM maven:3.5-jdk-8-alpine as build (2)
6  WORKDIR /app
7  COPY --from=clone /app/spring-petclinic /app (3)
8  RUN mvn install
9
10 FROM openjdk:8-jre-alpine
11 WORKDIR /app
12 COPY --from=build /app/target/spring-petclinic-1.5.1.jar /app
13 CMD ["java -jar spring-petclinic-1.5.1.jar"]
```

1	Labels the first stage as `clone`
2	Labels the second stage as `build`
3	References the first stage using the label

Choosing the Right Image(s)

Multiple-stage builds help tremendously with image size management, but it's not the only criterion. The image to start with has a big impact on the final image. Beginners generally use full-blown OS images, such as Ubuntu, inflating the size of the final image for no good reason. Yet, there are lightweight OSs that are very well-suited to Docker images, such as Alpine Linux.

It's also a great fit for security purposes, as the attack surface is limited.

In the above build file, the images are:

1. alpine/git
2. maven:3.5-jdk-8-alpine
3. openjdk:8-jre-alpine

They all inherit transitively or directly from Alpine.

Image sizes are as follows:

```
REPOSITORY                    TAG                 IMAGE ID            CREATED
nfrankel/spring-petclinic     latest              293bd333d60c        10 days ago
openjdk                       8-jre-alpine        c4f9d77cd2a1        4 weeks ago
```

The difference in size between the JRE and the app images is around 36 MB, which is the size of the JAR itself.

Exposing the Port

The Spring Pet Clinic is a web app, so it requires us to expose the HTTP port it will bind to. The relevant Docker directive is `EXPOSE`. I choose `8080` as a port number to be the same as the embedded Tomcat container, but it could be anything. The last stage should be modified as such:

```
FROM openjdk:8-jre-alpine
WORKDIR /app
COPY --from=build /app/target/spring-petclinic-1.5.1.jar /app
EXPOSE 8080
CMD ["java -jar spring-petclinic-1.5.1.jar"]
```

Parameterization

At this point, it appears the build file can be used for building *any* web app with the following features:

1. The source code is hosted on GitHub.
2. The build tool is Maven.
3. The resulting output is an executable JAR file.

Of course, that suits Spring Boot applications very well, but this is not a hard requirement.

Parameters include:

- The GitHub repository URL
- The project name

- Maven's `artifactId` and `version`
- The artifact name (as it might differ from the `artifactId`, depending on the specific Maven configuration)

Let's use those to design a parameterized build file. In Docker, parameters can be passed using either the `ENV` or `ARG` options. Both are set using the `--build-arg` option on the command-line. The differences between them are:

	`ENV`	`ARG`
Type		
Found in the image	Yes	No
Default value	Required	Optional

```
1  FROM alpine/git as clone
2  ARG url (1)
3  WORKDIR /app
4  RUN git clone ${url} (2)
5
6  FROM maven:3.5-jdk-8-alpine as build
7  ARG project (3)
8  WORKDIR /app
9  COPY --from=clone /app/${project} /app
10 RUN mvn install
11
12 FROM openjdk:8-jre-alpine
13 ARG artifactid
14 ARG version
15 ENV artifact ${artifactid}-${version}.jar (4)
16 WORKDIR /app
17 COPY --from=build /app/target/${artifact} /app
18 EXPOSE 8080
19 CMD ["java -jar ${artifact}"] (5)
```

1	`url` must be passed on the command line to set which GitHub repo to clone
2	`url` is replaced by the passed value
3	Same as <1>
4	`artifact` must be an `ENV`, so as to be persisted in the final app image
5	Use the `artifact` value *at runtime*

Building

The Spring Pet Clinic image can now be built using the following command-line:

```
1  docker build --build-arg url=https://github.com/spring-projects/spring-petclinic.git\
```

```
  --build-arg project=spring-petclinic\
  --build-arg artifactid=spring-petclinic\
  --build-arg version=1.5.1\
  -t nfrankel/spring-petclinic - < Dockerfile
```

Since the image doesn't depend on the filesystem, no context needs to be passed, and the Dockerfile can be piped from the standard input.

To build another app, parameters can be changed accordingly *e.g.*:

```
docker build --build-arg url=https://github.com/heroku/java-getting-started.git\
  --build-arg project=java-getting-started\
  --build-arg artifactid=java-getting-started\
  --build-arg version=1.0\
  -t nfrankel/java-getting-started - < Dockerfile
```

Running

Running an image built with the above command is quite easy:

```
docker run -ti -p8080:8080 nfrankel/spring-petclinic
```

Unfortunately, it fails with the following error message:

`starting container process caused "exec: \"java -jar ${artifact}\": executable file not found in $PATH.`

The trick is to use the `ENTRYPOINT` directive. The updated `Dockerfile` looks like the following:

```
FROM openjdk:8-jre-alpine
ARG artifactid
ARG version
ENV artifact ${artifactid}-${version}.jar (4)
WORKDIR /app
COPY --from=build /app/target/${artifact} /app
EXPOSE 8080
ENTRYPOINT ["sh", "-c"]
CMD ["java -jar ${artifact}"] (5)
```

At this point, running the container will (finally) work.

The second image uses a different port, so the command should be:

```
docker run -ti -p8080:5000 nfrankel/java-getting-started
```

Food for Thought

External Git

The current build clones a repository, and hence doesn't need to send the context to Docker. An alternative would be to clone outside the build file, *e.g.* in a continuous integration chain, and start from the context. That could be useful for building the image during development on developers machines.

Latest Version or Not?

For the Git image, I used the latest version, while for the JDK and JRE images, I used a specific major version. It's important for the Java version to be fixed to a major version — not so much for Git. It depends on the nature of the image.

Building From `master`

There's no configuration to change branches after cloning. This is wrong, as most of the time, builds are executed on a dedicated tag, *e.g.* `v1.0.0` . Obviously, there should be an additional command, as well as an additional build argument, to check out a specific tag before the build.

Skipping Tests

It takes an awful long time for the Pet Clinic application to build, as it has a huge test harness. Executing those tests takes time, and Maven must go through the `test` phase to reach the `install` phase. Depending on the specific continuous integration chain, tests might have been executed earlier and mustn't be executed again during the image build.

Maven Repository Download

Spring Boot apps have a lot of dependencies. It takes a long time for the image to build, as all dependencies need to be downloaded every time for every app. There are a couple of solutions to handle that. It probably will be the subject of another post.

The complete source code for this post can be found on GitHub.

To go further:

- Use multi-stage builds
- Dockerfile reference

Like This Article? Read More From DZone

Welcome to Spring Boot (With Embedded Tomcat Features)

Spring Boot Development With Docker

Multi-Stage Docker Image Build for

Free DZone Refcard

Java Apps

Getting Started With Vaadin Framework 8

Topics: DOCKER , MAVEN , SPRING BOOT , MULTI-STAGE , JAVA , TUTORIAL