**Spring Boot with Docker**

This guide walks you through the process of building a [Docker](https://docker.com/) image for running a Spring Boot application.

**What you’ll build**

[Docker](https://docker.com/) is a Linux container management toolkit with a "social" aspect, allowing users to publish container images and consume those published by others. A Docker image is a recipe for running a containerized process, and in this guide we will build one for a simple Spring boot application.

|  |  |
| --- | --- |
|  |  |

**What you’ll need**

* About 15 minutes
* A favorite text editor or IDE
* [JDK 1.8](http://www.oracle.com/technetwork/java/javase/downloads/index.html) or later
* [Maven 3.2+](https://maven.apache.org/download.cgi)
* You can also import the code straight into your IDE:
  + [Spring Tool Suite (STS)](https://spring.io/guides/gs/sts)

You will also need [Docker](https://docker.com/), which only runs on 64-bit machines. See <https://docs.docker.com/installation/#installation> for details on setting Docker up for your machine. Before proceeding further, verify you can run docker commands from the shell. If you are using boot2docker you need to run that **first**.

**Set up a Spring Boot app**

Now you can create a simple application.

src/main/java/hello/Application.java

package hello;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication

@RestController

public class Application {

@RequestMapping("/")

public String home() {

return "Hello Docker World";

}

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

The class is flagged as a @SpringBootApplication and as a @RestController, meaning it’s ready for use by Spring MVC to handle web requests. @RequestMapping maps / to the home() method which just sends a 'Hello World' response. The main() method uses Spring Boot’s SpringApplication.run() method to launch an application.

Now we can run the application without the Docker container (i.e. in the host OS).

If you are using Maven, execute:

./mvnw package && java -jar target/gs-spring-boot-docker-0.1.0.jar

and go to [localhost:8080](http://localhost:8080/) to see your "Hello Docker World" message.

**Containerize It**

Docker has a simple ["Dockerfile"](https://docs.docker.com/reference/builder/) file format that it uses to specify the "layers" of an image. So let’s go ahead and create a Dockerfile in our Spring Boot project:

Dockerfile

FROM openjdk:8-jdk-alpine

VOLUME /tmp

ARG JAR\_FILE

COPY ${JAR\_FILE} app.jar

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

This Dockerfile is very simple, but that’s all you need to run a Spring Boot app with no frills: just Java and a JAR file. The project JAR file is ADDed to the container as "app.jar" and then executed in the ENTRYPOINT.

|  |  |
| --- | --- |
|  | We added a VOLUME pointing to "/tmp" because that is where a Spring Boot application creates working directories for Tomcat by default. The effect is to create a temporary file on your host under "/var/lib/docker" and link it to the container under "/tmp". This step is optional for the simple app that we wrote here, but can be necessary for other Spring Boot applications if they need to actually write in the filesystem. |
|  | To reduce [Tomcat startup time](https://wiki.apache.org/tomcat/HowTo/FasterStartUp#Entropy_Source) we added a system property pointing to "/dev/urandom" as a source of entropy. This is not necessary with more recent versions of Spring Boot, if you use the "standard" version of Tomcat (or any other web server). |

To take advantage of the clean separation between dependencies and application resources in a Spring Boot fat jar file, we will use a slightly different implementation of the Dockerfile:

Dockerfile

FROM openjdk:8-jdk-alpine

VOLUME /tmp

ARG DEPENDENCY=target/dependency

COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib

COPY ${DEPENDENCY}/META-INF /app/META-INF

COPY ${DEPENDENCY}/BOOT-INF/classes /app

ENTRYPOINT ["java","-cp","app:app/lib/\*","hello.Application"]

This Dockerfile has a DEPENDENCY parameter pointing to a directory where we have unpacked the fat jar. If we get that right, it already contains a BOOT-INF/lib directory with the dependency jars in it, and a BOOT-INF/classes directory with the application classes in it. Notice that we are using the application’s own main class hello.Application (this is faster than using the indirection provided by the fat jar launcher).

|  |  |
| --- | --- |
|  | if you are using boot2docker you need to run it **first** before you do anything with the Docker command line or with the build tools (it runs a daemon process that handles the work for you in a virtual machine). |

To build the image you can use some tooling for Maven or Gradle from the community (big thanks to [Palantir](https://github.com/palantir/gradle-docker) and [Spotify](https://github.com/spotify/dockerfile-maven) for making those tools available).

**Build a Docker Image with Maven**

In the Maven pom.xml you should add a new plugin like this (see [the plugin documentation](https://github.com/spotify/dockerfile-maven) for more options): :

pom.xml

<properties>

<docker.image.prefix>springio</docker.image.prefix>

</properties>

<build>

<plugins>

<plugin>

<groupId>com.spotify</groupId>

<artifactId>dockerfile-maven-plugin</artifactId>

<version>1.4.9</version>

<configuration>

<repository>${docker.image.prefix}/${project.artifactId}</repository>

</configuration>

</plugin>

</plugins>

</build>

The configuration specifies 1 mandatory thing: the repository with the image name, which will end up here as springio/gs-spring-boot-docker.

Some other properties are optional:

* The name of the directory where the fat jar is going to be unpacked, exposing the Maven configuration as a build argument for docker. It can be specified using the <buildArgs/> of the plugin configuration.
* The image tag, which ends up as "latest" if not specified. It can be set with the <tag/>element,

|  |  |
| --- | --- |
|  | Before proceeding with the following steps (which use Docker’s CLI tools), make sure Docker is properly running by typing docker ps. If you get an error message, something may not be set up correctly. Using a Mac? Add $(boot2docker shellinit 2> /dev/null) to the bottom of your .bash\_profile (or similar env-setting configuration file) and refresh your shell to ensure proper environment variables are configured. |

To ensure the jar is unpacked before the docker image is created we add some configuration for the dependency plugin:

pom.xml

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<executions>

<execution>

<id>unpack</id>

<phase>package</phase>

<goals>

<goal>unpack</goal>

</goals>

<configuration>

<artifactItems>

<artifactItem>

<groupId>${project.groupId}</groupId>

<artifactId>${project.artifactId}</artifactId>

<version>${project.version}</version>

</artifactItem>

</artifactItems>

</configuration>

</execution>

</executions>

</plugin>

You can build a tagged docker image using the command line like this:

$ ./mvnw install dockerfile:build

And you can push the image to dockerhub with ./mvnw dockerfile:push.

|  |  |
| --- | --- |
|  | You don’t have to push your newly minted Docker image to actually run it. Moreover the "push" command will fail if you aren’t a member of the "springio" organization on Dockerhub. Change the build configuration and the command line to your own username instead of "springio" to make it actually work. |
|  | you can make dockerfile:push automatically run in the install or deploy lifecycle phases by adding it to the plugin configuration. |

pom.xml

<executions>

<execution>

<id>default</id>

<phase>install</phase>

<goals>

<goal>build</goal>

<goal>push</goal>

</goals>

</execution>

</executions>

**After the Push**

A "docker push" will fail for you (unless you are part of the organization at Dockerhub), but if you change the configuration to match your own docker ID then it should succeed, and you will have a new tagged, deployed image.

|  |  |
| --- | --- |
|  | You do NOT have to register with docker or publish anything to run a docker image. You still have a locally tagged image, and you can run it like this:  $ **docker run -p 8080:8080 -t springio/gs-spring-boot-docker**  ....  2015-03-31 13:25:48.035 INFO 1 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)  2015-03-31 13:25:48.037 INFO 1 --- [ main] hello.Application : Started Application in 5.613 seconds (JVM running for 7.293) |

The application is then available on [http://localhost:8080](http://localhost:8080/) (visit that and it says "Hello Docker World"). To make sure the whole process is really working, change the prefix from "springio" to something else (e.g. ${env.USER}) and go through it again from the build through to the docker run.

|  |  |
| --- | --- |
|  | When using a Mac with boot2docker, you typically see things like this at startup:  Docker client to the Docker daemon, please set:  export DOCKER\_CERT\_PATH=/Users/gturnquist/.boot2docker/certs/boot2docker-vm  export DOCKER\_TLS\_VERIFY=1  export DOCKER\_HOST=tcp://192.168.59.103:2376  To see the app, you must visit the IP address in DOCKER\_HOST instead of localhost. In this case, [http://192.168.59.103:8080](http://192.168.59.103:8080/), the public facing IP of the VM. |

When it is running you can see in the list of containers, e.g:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

81c723d22865 springio/gs-spring-boot-docker:latest "java -Djava.secur..." 34 seconds ago Up 33 seconds 0.0.0.0:8080->8080/tcp goofy\_brown

and to shut it down again you can docker stop with the container ID from the listing above (yours will be different):

$ docker stop 81c723d22865

81c723d22865

If you like you can also delete the container (it is persisted in your filesystem under /var/lib/docker somewhere) when you are finished with it:

$ docker rm 81c723d22865

**Using Spring Profiles**

Running your freshly minted Docker image with Spring profiles is as easy as passing an environment variable to the Docker run command

$ docker run -e "SPRING\_PROFILES\_ACTIVE=prod" -p 8080:8080 -t springio/gs-spring-boot-docker

or

$ docker run -e "SPRING\_PROFILES\_ACTIVE=dev" -p 8080:8080 -t springio/gs-spring-boot-docker

**Debugging the application in a Docker container**

To debug the application [JPDA Transport](https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/conninv.html#Invocation) can be used. So we’ll treat the container like a remote server. To enable this feature pass a java agent settings in JAVA\_OPTS variable and map agent’s port to localhost during a container run. With the [Docker for Mac](https://www.docker.com/products/docker" \l "/mac) there is limitation due to that we can’t access container by IP without [black magic usage](https://github.com/docker/for-mac/issues/171).

$ docker run -e "JAVA\_OPTS=-agentlib:jdwp=transport=dt\_socket,address=5005,server=y,suspend=n" -p 8080:8080 -p 5005:5005 -t springio/gs-spring-boot-docker

**Summary**

Congratulations! You’ve just created a Docker container for a Spring Boot app! Spring Boot apps run on port 8080 inside the container by default and we mapped that to the same port on the host using "-p" on the command line.