**Application Containerization using Docker**

Containers allow packaging of application together with libraries and other dependencies, providing isolated environments for running the software services.

Containers virtualize at the operating system level, with multiple containers running atop the OS kernel directly.

Docker is a popular, open-source Linux container format.

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Docker containers images become containers when they run on [Docker Engine](https://www.docker.com/products/container-runtime). Docker allows users to publish container images and consume those published by others.

Docker is supported by Kubernetes Engine, a tool that automates the deployment, management, scaling, networking, and availability of container-based applications.

This post will discuss the process of building docker image for running a Spring Boot using Windows Development Environment. We will use Kubernetes to deploy our application.

1. Create a Spring boot application
2. Install Virtual Linux server
3. Install Docker
4. Containerize the application
5. Install Kubernetes
6. Deploy the application to Kubernetes

**Create a Spring boot application**

@SpringBootApplication

@RestController

**public** **class** SpringbootApplication {

@RequestMapping("/welcome")

**public** String welcome() {

**return** "Example Application Home Page";

}

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringbootApplication.**class**, args);

}

}

Pom.xml: -

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>spring-docker</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>spring-docker</name>

<description>spring-docker</description>

<packaging>jar</packaging>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.3.3.RELEASE</version>

<relativePath />

</parent>

<properties>

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

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

Application.yml: -

server:

port: 9000

Execute the following command to build your application:

mvn package && java -jar target/spring-docker-0.0.1-SNAPSHOT.jar

Run and test your application localhost:9000/welcome.

Kill your application instance after testing.

**Install Virtual Linux server**

Here are the steps to install Virtual Linux server on Windows 10:

1. Ensure your machine is running Windows 10, updated to version 2004, Build 18362 or higher.
2. Enable the Windows Subsystem for Linux. Open PowerShell as Administrator and run:

dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart

With the WSL 2 backend supported in Docker Desktop for Windows, you can work in a Linux-based development environment and build Linux-based containers.

1. Install Linux distribution (e.g. Ubuntu).
   1. From the distribution's page, select "Get".
   2. Create a user account and password for your new Linux distribution.
2. Install Java on Linux server

apt install default-jdk

1. Install Maven on Linux server

apt install maven

**Install Docker**

1. Download [Docker Desktop](https://docs.docker.com/docker-for-windows/wsl/#download) and follow the installation instructions.
2. Once installed, start Docker Desktop from the Windows Start menu, then select the Docker icon from the hidden icons menu of your taskbar. Right-click the icon to display the Docker commands menu and select "Settings".
3. Ensure that "Use the WSL 2 based engine" is checked in **Settings** > **General**.
4. Select from your installed WSL 2 distributions which you want to enable Docker integration on by going to: **Settings** > **Resources** > **WSL Integration**.
5. To confirm that Docker has been installed, open your WSL distribution and display the version and build number by entering: docker --version
6. Test that your installation works correctly by running a simple built-in Docker image using: docker run hello-world
7. Create you Dockerhub ID.

**Containerize the application**

1. Create a Dockerfile in our Spring Boot project, which lists the components and commands that make up the package.

FROM openjdk:8-jdk-alpine

ARG JAR\_FILE=target/\*.jar

COPY ${JAR\_FILE} app.jar

ENTRYPOINT ["java","-jar","/app.jar"]

We need Java and a JAR file to run this Spring Boot app. The build will create a spring user and a spring group to run the application. It will then COPY the project JAR file into the container as "app.jar" that will be executed in the ENTRYPOINT.

1. We can run it using Maven with

$ docker build -t spring-docker.

This command builds an image and tags it as spring-docker.

1. We can also build a Docker Image with Maven using Spring Boot image generator (Dockerfile will be ignored): $ mvn spring-boot:build-image -Dspring-boot.build-image.imageName=<Dockerhub ID>/spring-docker
2. Run this command to list the images in the repository: $ docker images
3. Run the docker container: docker run -p 9000:9000 -t <Dockerhub ID>/spring-docker
4. Test your application localhost:9000/welcome.

6. Kill the container instance after testing your application.

1. Login to Dockerhub: $ docker login
2. Push the image to Dockerhub: docker push <Dockerhub ID>/spring-docker.
3. The image needs to be pushed to an accessible registry.( Kubernetes pulls the image from inside its Kubelets (nodes), which are not in general connected to the local docker daemon.) We will discuss creating local registry in the subsequent post.

**Install Kubernetes**

1. Select the Docker icon from the menu of your taskbar. Right-click the icon to display the Docker commands menu and select “Settings”.
2. Check "Enable Kubernetes" under Kubernetes. It will take few minutes to start Kubernetes cluster.
3. Check that you have a Kubernetes cluster running by executing command:

kubectl get all

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

**Deploy the application to Kubernetes**

1. Run the following commands to generate deployment properties file for Kubernetes: -

kubectl create deployment spring-docker --image=asmanaeem/spring-docker --dry-run=client -o=yaml > deployment.yaml

echo --- >> deployment.yaml

kubectl create service clusterip spring-docker --tcp=9000:9000 --dry-run=client -o=yaml >> deployment.yaml

1. Apply the generated YAML file:

kubectl apply -f deployment.yaml

deployment.apps/ spring-docker created

service/ spring-docker created

1. Check that the application is running:

kubectl get all

|  |
| --- |
| NAME READY STATUS RESTARTS AGE |
| pod/spring-docker-68fd9fbc96-wbv8v 1/1 Running 0 17s |
|  |
| NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE |
| service/kubernetes ClusterIP 10.97.0.1 <none> 443/TCP 5m29s |
| service/spring-docker ClusterIP 10.104.2.156 <none> 9000/TCP 17s |
|  |
| NAME READY UP-TO-DATE AVAILABLE AGE |
| deployment.apps/spring-docker 1/1 1 1 17s |
|  |
| NAME DESIRED CURRENT READY AGE |
| replicaset.apps/spring-docker-68fd9fbc96 1 1 1 17 |

1. Now you need to be able to connect to the application, which you have exposed as a Service in Kubernetes. One way to do that, which works great at development time, is to create an SSH tunnel:

kubectl port-forward service/spring-docker 9000:9000

The application is then available on http://localhost:9000/welcome (visit that and it says "Example Application Home Page!"}

**Externalize configuration using Spring Cloud Config: -**

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system.

Configurations can be pulled from a git repository, local filesystems, Subversion, Hashicorp Vault, JDBC compatible database, and Credhub.

Set up a Spring Cloud Config Service

1. Dependencies:

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.9.RELEASE</version>

<relativePath />

</parent>

<properties>

<docker.image.prefix>example</docker.image.prefix>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

</dependencies>

1. Spring Boot Application: -

Use @EnableConfigServer annotation to setup the application.

@EnableConfigServer

@SpringBootApplication

**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(Application.**class**, args);

}

}

1. Configure the repository URL in the application.properties

server.port=8888

spring.cloud.config.server.git.uri=https://github.com/mygit/spring-cloud-config-repo

health.config.enabled=false

The HTTP service has resources in the following form:

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

where application is injected as the spring.application.name in the Spring Boot Application, profile is an active profile (or comma-separated list of properties), and label is an optional git label (defaults to master.)

Source Code

Client-Side Usage

1. To use these features in an application, we can build it as a Spring Boot application that depends on spring-cloud-config-client.

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.9.RELEASE</version>

<relativePath />

</parent>

<properties>

<docker.image.prefix>example</docker.image.prefix>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

</dependencies>

</build>

<!-- repositories also needed for snapshots and milestones -->

1. Create a standard Spring Boot application, such as the following HTTP server:

@SpringBootApplication

**public** **class** SpringbootApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(SpringbootApplication.**class**, args);

}

@RefreshScope

@RestController

**class** MessageRestController {

@Value("${message:Example Application Home Page!}")

**private** String message;

@RequestMapping("/welcome")

String getMessage() {

**return** **this**.message;

}

}

}

When this HTTP server runs, it picks up the external configuration from the default local config server (if it is running) on port 8888. To modify the startup behavior, we can change the location of the config server by using bootstrap.properties (similar to application.properties but for the bootstrap phase of an application context), as in our example:

server.port=9001

spring.application.name=spring-cloud-config-client0

#spring.profiles.active=production

spring.cloud.config.uri=http://192.168.1.152:8888

management.security.enabled=false

health.config.enabled=false

logging.level.=DEBUG

logging.file=c:/tmp/app.log

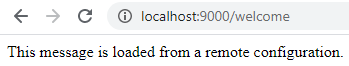
By default, if spring.application.name is not set, “application” will be used.

With the variable spring.profiles.active, we indicate which profile will be used. If we do not put any (as is the case for our commented line), the profile used will be the default.

1. Create your configuration file and push it to the repository.

spring-cloud-config-client0.properties: -

message=This message is loaded from a remote configuration.



Source Code