Spring Boot: Docker Image

Dieser Artikel beschreibt am Beispiel des Spring-Boot Erueka Discovery Server, wie ein Spring Boot Server in ein Docker Image verpackt und gestartet wird. Vorausgesetzt wird eine Java 1.8 Installation, eine betriebsbereite Docker Installation und eine Java IDE bzw. ein guter Editor. Ein Build-Tool wie Maven oder Gradle ist entweder Bestandteil der IDE oder installiert.
Die Beispiele wurden mit der Docker Engine 20.10.5, der Eclipse-IDE 2021-03 und dem Java 1.8.0_271 auf einem MacBook Air erstellt. Die Eclipse-IDE wurde um die Module Eclipse-Docker-Tooling 5.2 und Spring-Tools 4 aus dem Eclipse Marketplace erweitert.

Das Beispiel nimmt keine Rücksicht auf Sicherheitsaspekte wie z. B. Spring Boot Security, sichere Protokolle wie ssl, OAuth2 etc. Es soll nur beispielhaft gezeigt werden, welche grundsätzlichen Schritte notwendig sind, um einen Spring Boot Server in ein Docker Image zu packen.

Anlegen eines Spring-Boot Projektes

Ein Spring Boot Projekt mit dem Eureka Discovery Server wird über *Projekte/Spring Boot/Spring Starter Project* erzeugt. Im Projektdialog legt man auf der ersten Seite Projektname und Group, Artefact-ID und weiter Grundeinstellungen für das Projekt fest. Der Projekttyp ist Maven, das Packaging jar, die Java-Version 8 und die Sprache Java.

Auf der zweiten Seite des Dialogs werden der Eureka Server und Spring Boot Actuator ausgewählt. Sie können bequem über die Suchfunktion durch Eingabe von Eureka und Actuator gefunden werden. Mit Finish wird anschließend das Projekt erstellt.

Wer nicht über eine entsprechende Erweiterung in seiner IDE verfügt, kann alternativ den Spring Initializer auf https://start.spring.io verwenden. Die Vorgehensweise ist ganz ähnlich. Die erzeugte ZIP-Datei enthält die benötigte Projektstruktur.

Anpassungen im Projekt

Um den Eureka-Server zu aktivieren ist die Main-Class um die Annotation @EnableEurekaServer zu erweitern. Die Annotation muss zusätzlich importiert werden:

```
package de.rastef.jusoko.esrserver;


import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
```

```
@SpringBootApplication
@EnableEurekaServer
public class JusokoEsrserverApplication {

    public static void main(String[] args) {
        SpringApplication.run(JusokoEsrserverApplication.class, args);
    }
```

Zusätzlich sind in den Application Properties neue Properties einzutragen:

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
management.endpoints.web.exposure.include=info,health,env
```

In der ersten Zeile weisen wir dem Server ein Port zu. Über http://localhost:8761/ ist nach dem Start des Servers dessen Dashboard erreichbar. Die zwei folgenden Zeilen, verhindern, dass sich der Server selbst bei anderen Eureka-Servern zu registrieren versucht. Minder letzten Zeile werden in der Actuator-Schnittstelle die Funktionen info, health und env für das Webinterface freigegeben. Standardmäßig sind nur die ersten beiden Funktionen über das Web erreichbar. Eine Actuatorfunktion wird über die URL http//localhost:8761/actuator/<function> aufgerufen. Für die Health-Funktion lautet der Aufruf also http://localhost:8761/actuator/health.

Nach diesen Änderungen kann das Projekt mit maven clean und maven install übersetzt werden. Ein erster Start erfolgt in Eclipse mit *Run As Spring Boot App*. Die zuvor genannten URLs sollten nun erreichbar sein.

Dockerize it

Um den Server in einen Docker-Container zu packen, muss die Datei Dockerfile im Rootverzeichnis des Projektes angelegt werden. Die Datei erhält folgenden Inhalt:

```
FROM openjdk:8-jdk-alpine
RUN addgroup -S esrsrvr && adduser -S esrsrvr -G esrsrvr
USER esrsrvr:esrsrvr
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Das Image benötig eine Java Laufzeitumgebung. Für den Bau des Image wird das schlanke Alpine-JDK genutzt. Da der eigentliche Server nicht als root laufen soll, wird eine Gruppe esrsrvr und ein User esrsrvr angelegt, der der Gruppe esrsrvr zugewiesen wird. Unter diesem User und dieser Gruppe wird dann der Server gestartet. Dazu wird die durch den Build erzeugte Datei target/app.jar in das Image kopiert und der entsprechende Entrypoint gesetzt. Mit

```
docker build -t rastef/esr-service .
```

wir nun das Image gebaut. In diesem Fall heißt das Image rastef/esr-service.

Docker Image starten

Das erste mal wird das Image mit dem Befehl

```
docker run -p 8761:8761 -e SPRING_PROFILES_ACTIVE=prod,actuator --name esr rastef/esr-service
```

gestartet. Der von Docker erzeugte Container erhält dabei den Namen esr. Über diesen ist er bei späteren Start-/Stop-Kommandos einfacher anzusprechen. Die URLs aus dem obigen Test sollten nun genauso funktionieren. Angehalten wird der Container mit STRG+C.

Der nächste Start desselben Container kann mit

```
docker start esr
```

erfolgen. Angehalten wird er mit

```
docker stop esr
```

Wird der Container nicht mehr benötigt, sollte er mit

```
docker rm esr
```

entfernt werden.

Revision #1
Created 2026-09-26 13:20:53 UTC by ralf
Updated 2026-09-26 13:20:53 UTC by ralf