**CI/CD Deployment for Spring Boot Application**

**Writeup:**

We'll create a simple Spring Boot application and then push it to the shared Git repository. After that, we'll build it with a building integration service, create a Docker image, and push it to a Docker repository.

In the end, we'll automatically deploy our application to a PaaS service (Heroku).

2. Version Control

The crucial part of CI/CD is the version control system to manage our code. In addition, we need a repository hosting service that our build and deploy steps will tie into.

Let's choose Git as the VCS and GitHub as our repository provider as they are the most popular at the moment and free to use.

First, we'll have to create an account on GitHub.

Additionally, we should create a Git repository. Let's name it baeldung-ci-cd-process. Also, let's pick a public repository since it will allow us to access other services for free.

Now that our repository has been created, we should clone our project locally. To do so, let's execute this command on our local computer:

3. Creating the Application

In this section, we'll create a simple Spring Boot application that will take part in the process. We'll also use Maven as our build tool.

First, let's initialize our project in the directory where we have cloned the version control repository.

For instance, we can do that with Spring Initializer, adding the web and actuator modules.

3.1. Creating the Application Manually

Or, we can add the spring-boot-starter-web and spring-boot-starter-actuator dependencies manually:

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

The first is to introduce a REST endpoint and the second, a health check endpoint.

In addition, let's add the plugin that'll allow us to run our application:

<plugin>

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

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

</plugin>

And finally, let's add a Spring Boot main class:

@SpringBootApplication

public class CiCdApplication {

public static void main(String[] args) {

(CiCdApplication.class, args);

}

}

3.2. Pushing

Whether using Spring Initializr or manually creating the project, we're now ready to commit and push our changes to our repository.

Let's do that with the following commands:

git add .

git commit -m 'Initialize application'

git push

We can now check if our changes exist in the repository.

4. Build Automation

Another part of the CI/CD process is a service that will build and test our pushed code.

We'll use Travis CI here, but any building service will work as well.

4.1. Maven Wrapper

Let's begin by adding a Maven Wrapper to our application. If we've used the Spring Initializr, we can skip this part since it's included by default.

In the application directory, let's do:

mvn -N io.takari:maven:0.7.7:wrapper

This will add Maven wrapper files, including the mvnw and the mvnw.cmd files that can be used instead of Maven.

While Travis CI has its own Maven, other building services might not. This Maven Wrapper will help us to be prepared for either situation. Also, developers won't have to install Maven on their machines.

4.2. Building Service

After that, let's create an account on Travis CI by signing in with our GitHub account. Going forward, we should allow access to our project in GitHub.

Next, we should create a .travis.yml file which will describe the building process in Travis CI. Most of the building services allow us to create such a file, which is located at our repository's root.

In our case, let's tell Travis to use Java 11 and the Maven Wrapper to build our application:

language: java

jdk:

- openjdk11

script:

- ./mvnw clean install

The language property indicates we want to use Java.

The jdk property says which Docker image to download from DockerHub, openjdk11 in this case.

The script property says what command to run – we want to use our Maven wrapper.

freestar

Lastly, we should push our changes to the repository. The Travis CI should automatically trigger the build.

5. Dockerizing

In this section, we'll build a Docker image with our application and host it on DockerHub as a part of the CD process. It'll allow us to run it on any machine with ease.

5.1. Repository for Docker Images

First, we should create a Docker repository for our images.

Let's create an account on DockerHub. Also, let's create the repository for our project by filling out the appropriate fields:

name: baeldung-ci-cd-process

visibility: Public

Build setting: GitHub

5.2. Docker Image

Now, we're ready to create a Docker image and push it to DockerHub.

First, let's add the jib-maven-plugin that will create and push our image with the application to the Docker repository (replace DockerHubUsername with the correct username):

<profile>

<id>deploy-docker</id>

<properties>

<maven.deploy.skip>true</maven.deploy.skip>

</properties>

<build>

<plugins>

<plugin>

<groupId>com.google.cloud.tools</groupId>

<artifactId>jib-maven-plugin</artifactId>

<version>2.2.0</version>

<configuration>

<to>

<image>${DockerHubUsername}/baeldung-ci-cd-process</image>

<tags>

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

<tag>latest</tag>

</tags>

</to>

</configuration>

</plugin>

</plugins>

</build>

</profile>

We have added it as part of a Maven profile in order to not trigger it with the default build.

Additionally, we have specified two tags for the image. To learn more about the plugin, visit our article about Jib.

Going forward, let's adjust our build file (.travis.yml):

before\_install:

- echo "$DOCKER\_PASSWORD" | docker login -u "$DOCKER\_USERNAME" --password-stdin

- docker pull openjdk:11-jre-slim-sid

script:

- ./mvnw clean install

- ./mvnw deploy jib:build -P deploy-docker

With these changes, the build service will log in to DockerHub before building the application. Additionally, it'll execute the deploy phase with our profile. During that phase, our application will be pushed as an image to the Docker repository.

Lastly, we should define DOCKER\_PASSWORD and DOCKER\_USERNAME variables in our build service. In Travis CI, these variables can be defined as part of the build settings.

Now, let's push our changes to the VCS. The build service should automatically trigger the build with our changes.

We can check if the Docker image has been pushed to the repository by running locally:

docker run -p 8080:8080 -t $DOCKER\_USERNAME/baeldung-ci-cd-process

Now, we should be able to access our health check by accessing http://localhost:8080/actuator/health.

6. Code Analysis

The next thing we'll include in our CI/CD process is static code analysis. The main goal of such a process is to ensure the highest code quality. For instance, it could detect that we don't have enough test cases or that we have some security issues.

Let's integrate with CodeCov, which will inform us about our test coverage.

Firstly, we should log in to CodeCov with our GitHub profile to establish integration.

Secondly, we should make changes to our code. Let's begin by adding the jacoco plugin:

<plugin>

<groupId>org.jacoco</groupId>

<artifactId>jacoco-maven-plugin</artifactId>

<version>0.8.5</version>

<executions>

<execution>

<id>default-prepare-agent</id>

<goals>

<goal>prepare-agent</goal>

</goals>

</execution>

<execution>

<id>report</id>

<phase>test</phase>

<goals>

<goal>report</goal>

</goals>

</execution>

</executions>

</plugin>

The plugin is responsible for generating test reports that will be used by CodeCov.

Next, we should adjust the script section in our build service file (.travis.yml):

script:

- ./mvnw clean org.jacoco:jacoco-maven-plugin:prepare-agent install

- ./mvnw deploy jib:build -P deploy-docker

after\_success:

- bash <(curl -s https://codecov.io/bash)

We instructed the jacoco plugin to trigger during the clean install phase. Additionally, we've included the after\_success section, which will send the report to CodeCov after the build is successful.

Going forward, we should add a test class to our application. For instance, it could be a test for the main class:

@SpringBootTest

class CiCdApplicationIntegrationTest {

@Test

public void contextLoads() {

}

}

Lastly, we should push our changes. The build should be triggered and the report should be generated in our CodeCov profile related to the repository.

7. Deploying the Application

As the last part of our process, we'll deploy our application. With a Docker image available for use, we can deploy it on any service. For instance, we could deploy it on cloud-based PaaS or IaaS.

Let's deploy our application to Heroku, which is a PaaS that requires minimal setup.

First, we should create an account and then log in.

Next, let's create the application space in Heroku and name it baeldung-ci-cd-process. The name of the application must be unique, so we might need to use another one.

We'll deploy it by integrating Heroku with GitHub, as it's the simplest solution. However, we could also have written the pipeline that would use our Docker image.

Going forward, we should include the heroku plugin in our pom:

<profile>

<id>deploy-heroku</id>

<properties>

<maven.deploy.skip>true</maven.deploy.skip>

</properties>

<build>

<plugins>

<plugin>

<groupId>com.heroku.sdk</groupId>

<artifactId>heroku-maven-plugin</artifactId>

<version>3.0.2</version>

<configuration>

<appName>spring-boot-ci-cd</appName>

<processTypes>

<web>java $JAVA\_OPTS -jar -Dserver.port=$PORT target/${project.build.finalName}.jar</web>

</processTypes>

</configuration>

</plugin>

</plugins>

</build>

</profile>

Like with Docker, we've added it as part of the Maven profile. Additionally, we've included a startup command in the web section.

Next, we should adjust our build service file (.travis.yml) to deploy the application to Heroku as well:

script:

- ./mvnw clean install

- ./mvnw heroku:deploy jib:build -P deploy-heroku,deploy-docker

Additionally, let's add the Heroku API-KEY in a HEROKU\_API\_KEY variable in our building service.

Lastly, let's commit our changes. The application should be deployed to Heroku after the build has finished.