DISEÑO E IMPLEMENTACIÓN DE CI/CD

Marc Peligros Fort

LA SALLE CAMPUS BCN 14/12/2021

Contenido

1. Introducción

Para la realización de la siguiente actividad se nos requería disponer de un sistema de integración y distribución continuas, ficticio ya que solo se requería que se mostraran echos por pantalla, pero los comandos con cierto sentido y que nos hubieran funcionado en local antes.

Es decir, era necesario pensar y plantear un sistema CI/CD que se acercara lo mas posible a uno funcional.

Para ello el primer paso necesario era disponer de una aplicación simple con la que poder hacer las diferentes pruebas, dockerizarla, ponerla en kubernetes para así poder realizar dicho sistema.

2. La aplicación

Para la aplicación se decidió realizar en Java ya que era un lenguaje que estaba acostumbrado mucho a programar en mi trabajo pero en una plataforma totalmente monolita, quería ver una aplicación Java en un entorno de Docker/kubernetes para poder visualizar-la mejor en un entorno así.

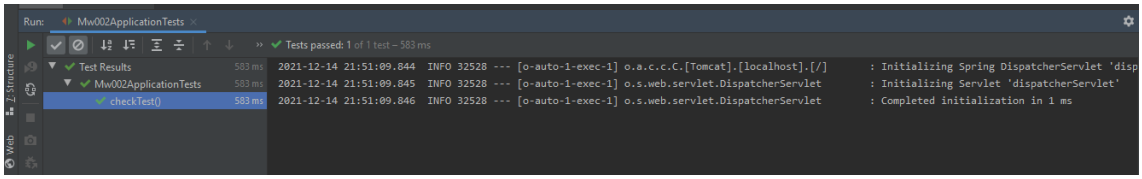
```java
package com.peligros.test.mw002;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;


@SpringBootApplication
@RestController
public class Mw002Application extends SpringBootServletInitializer {

    public static void main(String[] args) { SpringApplication.run(Mw002Application.class, args); }

    @RequestMapping("/")
    public String home() { return "Hello world"; }

}
```

Para ello se decidió crear un proyecto con Spring boot + Maven + JDK8 para crear un hello world de lo mas simple.

Junto a la siguiente prueba:

```java
package com.peligros.test.mw002;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.web.server.LocalServerPort;

import static org.assertj.core.api.Assertions.assertThat;
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class Mw002ApplicationTests {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void checkTest() throws Exception {
        assertThat(this.restTemplate.getForObject( url: "http://localhost:" + port + "/",
                String.class)).contains("Hello world");
    }
}
```

De esta manera si se lanzaba la prueba y la página web contenía hello world la daba por buena.



Tuve problemas a la hora de poder llevarme el programa a Docker ya que quería hacerlo creando un .jar y ejecutándolo en una imagen llamada: openjdk:8-jdk-alpine https://hub.docker.com/layers/openjdk/library/openjdk/8-jdk-alpine/images/sha256-a3562aa0b991a80cfe8172847c8be6dbf6e46340b759c2b782f8b8be45342717?context=explore (ya que lo que quería era una imagen con lo muy mínimo para poder ejecutar una aplicación con jdk 8).

El problema era que el .JAR que creaba con InteliJ daba problemas en el Docker, no sabía muy bien porque pero con InteliJ funcionaba pero en mi consola de comandos no, era que por algún motivo no creaba el .JAR con las dependencias correctas, después de pelearme durante un rato ví que también se podía compilar con el propio Maven con un 'mvn clean package'

De esta manera se creaba un .JAR que no solo podía ejecutar desde mi consola de comandos si no también desde el propio Docker.

3. Prueba con Docker

Primero de todo realicé una prueba con Docker ya que si con Docker no me funcionaba entonces no valía la pena subir la imagen al Dockerhub, así que cree un Dockerfile sencillo para poder realizar la prueba, donde utilizaba el puerto 8082 que no tenía conflicto con ningún servicio de mi ordenador y me dispuse a probarlo:

```
C:¥Users¥dangerscito¥Desktop¥Docker¥final>docker run -d -p 8082:8082 --name test topmasses/spring-hello-world:0.0.2
712e9543e409604c8e91b0a58b0e6833b194f5a0a3a5fcd187a3c91b10b1c7a8

C:¥Users¥dangerscito¥Desktop¥Docker¥final>docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED          STATUS          PORTS                                                  NAMES
712e9543e409   topmasses/spring-hello-world:0.0.2   "java -Xmx750m -jar …"   3 seconds ago    Up 2 seconds    0.0.0.0:8082->8082/tcp                                 test
f0bb2e53a31a   jenkins/jenkins:lts-jdk11            "/sbin/tini -- /usr/…"   3 days ago       Up 2 days       0.0.0.0:8080->8080/tcp, 0.0.0.0:50000->50000/tcp       trusting_bardeen

C:¥Users¥dangerscito¥Desktop¥Docker¥final>
```

Hello world

Una vez realizado decidí desplegar la imagen en mi cuenta de Dockerhub, como se puede ver en la siguiente imagen:

```
C:¥Users¥dangerscito>docker login --username topmasses
Password:
Login Succeeded

Logging in with your password grants your terminal complete access to your accou
nt.
For better security, log in with a limited-privilege personal access token. Lear
n more at https://docs.docker.com/go/access-tokens/

C:¥Users¥dangerscito>docker push topmasses/spring-hello-world
Using default tag: latest
The push refers to repository [docker.io/topmasses/spring-hello-world]
8cc4131b1246: Pushing  11.21MB/17.62MB
ceaf9e1ebef5: Mounted from library/openjdk
9b9b7f3d56a0: Mounted from library/openjdk
f1b5933fe4b5: Mounted from library/openjdk
```

Inmediatamente después me di cuenta de que la había subido sin tag, realizando un Docker tag pude renombrar la imagen y subirla en versión 0.0.1 (posteriormente subí una 0.0.2)

https://hub.docker.com/r/topmasses/spring-hello-world/tags

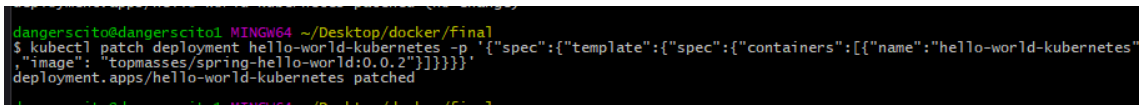
Una vez con la imagen subida podía pasar a la parte de kubernetes realizada con Minikube:

Para ello se hizo un kubectl apply -f de los dos objetos de la carpeta kubernetes_manifest.

Una era un nodeport simple que apuntaba a un objecto deployment, y el otro un deployment que tenía tres replicas y tenía un 25% mínimo de disponibilidad y un 25% de replicas adicionales en caso de necesidad, para así poder garantizar en despliegues o cambios que el usuario va a poder acceder.

Como era un nodeport me conecté a la IP de minikube ip que me lanzaba para poder probar que realmente funcionaba en un entorno kubernetes.

Aprovechando que estaba en un entorno kubernetes pude probar la que seria la comanda que solucionaría el problema a la hora de hacer un deploy en kubernetes, el cambio de imagen:



4. Confección del jenkinsfile

La configuración de Jenkins quedará en un documento PDF aparte en instrucciones de la configuración de Jenkins (carpeta jenkins_instructions)

Ver carpeta jenkinsfile para poder ver el archivo del que se hablara.

Para definir la pipeline se establecieron las 'tools' o herramientas, Maven 3.8.4 y jdk8, que habían sido las herramientas para instalar el proyecto, así una vez ejecutado el jenkinsfile podría hacer uso del compilador de Maven para poder crear jars.

En la primera stage, la de build and test, se juntaron en una porque Maven a la hora de realizar su instrucción mvn clean package también ejecutaba las pruebas, lanzando una excepción en caso de fallara a no ser que se pusiera con el flag *-Dmaven.test.failure.ignore=true.*

En este caso me interesaba que lanzara excepción así pudiendo lanzar un error y cortar la pipeline en caso de que el test no fuera bien, lanzando un error con Jenkins con la instrucción err, lanzando antes la excepción por consola para que se pueda ver en los logs.

En caso de que funcionara, se creaban los xml con los reports y además, se archivaba el jar creado con archiveArtifacts de Jenkins, me pareció interesante pues permitía poder descargar los propios jar desde Jenkins así poder tener un sitio de backup para poder descargar los mismos.

Después defino una variable que será indispensable para la parte de delivery y deployment pues va a definir la versión o tag del proyecto.

sh script: 'mvn help:evaluate -Dexpression=project.version -q -DforceStdout', returnStdout: true

Con esta instrucción ejecutada desde la raíz del proyecto Java, permite obtener la versión de la aplicación, pudiendo hacer que la versión de la imagen sea la misma que de la aplicación, así ahorrando posibles complicaciones.

Para la parte de delivery se decide hacer simple, solo actualizando la imagen en dockerhub (indispensable para el deployment) y también actualizar la rama master del git.

De esta manera en la parte de deployment con un simple patch cambiando el numero de la versión al nuevo, conseguiríamos actualizarlo, debido a que el deployment que tenemos configurado, hará el cambio gradual, apagando replicas para actualizarlas a la nueva versión sin dejar de ofrecer al menos 1 en pie, de esta manera garantizando que la aplicación siga funcionando.