**Trabajo Práctico 9 - Pruebas de unidad**

1- Objetivos de Aprendizaje

- Adquirir conocimientos sobre conceptos referidos a pruebas de unidad (unit tests).

- Generar y ejecutar pruebas unitarias utilizado frameworks disponibles.

2- Unidad temática que incluye este trabajo práctico

Este trabajo práctico corresponde a la unidad Nº: 5 (Libro Ingeniería de Software: Cap 8)

3- Consignas a desarrollar en el trabajo práctico:

Conceptos generales explicaciones de los mismos

¿Qué son las pruebas de software?

Una prueba de software es una pieza de software que ejecuta otra pieza de software. Valida si ese código da como resultado el estado esperado (prueba de estado) o ejecuta la secuencia de eventos esperados (prueba de comportamiento).

¿Por qué son útiles las pruebas de software?

Las pruebas de la unidad de software ayudan al desarrollador a verificar que la lógica de una parte del programa sea correcta.

Ejecutar pruebas automáticamente ayuda a identificar regresiones de software introducidas por cambios en el código fuente. Tener una cobertura de prueba alta de su código le permite continuar desarrollando características sin tener que realizar muchas pruebas manuales.

Código (o aplicación) bajo prueba

El código que se prueba generalmente se llama código bajo prueba . Si está probando una aplicación, esto se llama la aplicación bajo prueba .

Prueba unitarias (Unit Tests)

Una prueba de unidad es una pieza de código escrita por un desarrollador que ejecuta una funcionalidad específica en el código que se probará y afirma cierto comportamiento o estado.

El porcentaje de código que se prueba mediante pruebas unitarias generalmente se llama cobertura de prueba.

Una prueba unitaria se dirige a una pequeña unidad de código, por ejemplo, un método o una clase. Las dependencias externas deben eliminarse de las pruebas unitarias, por ejemplo, reemplazando la dependencia con una implementación de prueba o un objeto (mock) creado por un framework de prueba.

Las pruebas unitarias no son adecuadas para probar la interfaz de usuario compleja o la interacción de componentes. Para esto, debes desarrollar pruebas de integración.

Frameworks de pruebas unitarias para Java

Hay varios frameworks de prueba disponibles para Java. Los más populares son JUnit y TestNG

¿Qué parte del software debería probarse?

Lo que debe probarse es un tema muy controvertido. Algunos desarrolladores creen que cada declaración en su código debe ser probada.

En cualquier caso, debe escribir pruebas de software para las partes críticas y complejas de su aplicación. Si introduce nuevas funciones, un banco de pruebas sólido también lo protege contra la regresión en el código existente.

En general, es seguro ignorar el código trivial. Por ejemplo, es inútil escribir pruebas para los métodos getter y setter que simplemente asignan valores a los campos. Escribir pruebas para estas afirmaciones consume mucho tiempo y no tiene sentido, ya que estaría probando la máquina virtual Java. La propia JVM ya tiene casos de prueba para esto. Si está desarrollando aplicaciones de usuario final, puede suponer que una asignación de campo funciona en Java.

Si comienza a desarrollar pruebas para una base de código existente sin ninguna prueba, es una buena práctica comenzar a escribir pruebas para el código en el que la mayoría de los errores ocurrieron en el pasado. De esta manera puede enfocarse en las partes críticas de su aplicación.

4- Desarrollo:

**1- Familiarizarse con algunos conceptos del framework JUnit:**

|  |  |
| --- | --- |
| JUnit 4 | Descripción |
| import org.junit.\* | Instrucción de importación para usar las siguientes anotaciones. |
| @Test | Identifica un método como un método de prueba |
| @Before | Ejecutado antes de cada prueba. Se utiliza para preparar el entorno de prueba (por ejemplo, leer datos de entrada, inicializar la clase). |
| @After | Ejecutado después de cada prueba. Se utiliza para limpiar el entorno de prueba (por ejemplo, eliminar datos temporales, restablecer los valores predeterminados). También puede ahorrar memoria limpiando costosas estructuras de memoria. |
| @BeforeClass | Ejecutado una vez, antes del comienzo de todas las pruebas. Se usa para realizar actividades intensivas de tiempo, por ejemplo, para conectarse a una base de datos. Los métodos marcados con esta anotación deben definirse static para que funcionen con JUnit |
| @AfterClass | Ejecutado una vez, después de que se hayan terminado todas las pruebas. Se utiliza para realizar actividades de limpieza, por ejemplo, para desconectarse de una base de datos. Los métodos anotados con esta anotación deben definirse static para que funcionen con JUnit. |
| @Ignore o @Ignore("Why disabled") | Marca que la prueba debe ser deshabilitada. Esto es útil cuando se ha cambiado el código subyacente y el caso de prueba aún no se ha adaptado. O si el tiempo de ejecución de esta prueba es demasiado largo para ser incluido. Es una mejor práctica proporcionar la descripción opcional, por qué la prueba está deshabilitada. |
| @Test (expected = Exception.class) | Falla si el método no arroja la excepción nombrada. |
| @Test(timeout=100 | Falla si el método tarda más de 100 milisegundos. |

|  |  |
| --- | --- |
| Declaración | Descripción |
| fail ([mensaje]) | Deja que el método falle. Se puede usar para verificar que no se llegue a una determinada parte del código o para realizar una prueba de falla antes de implementar el código de prueba. El parámetro del mensaje es opcional. |
| assertTrue ([mensaje,] condición booleana) | Comprueba que la condición booleana es verdadera. |
| assertFalse ([mensaje,] condición booleana) | Comprueba que la condición booleana es falsa. |
| assertEquals ([mensaje,] esperado, real) | Comprueba que dos valores son iguales. Nota: para las matrices, la referencia no se verifica en el contenido de las matrices. |
| assertEquals ([mensaje,] esperado, real, tolerancia) | Pruebe que los valores float o double coincidan. La tolerancia es el número de decimales que debe ser el mismo. |
| assertNull (objeto [mensaje,]) | Verifica que el objeto sea nulo. |
| assertNotNull (objeto [mensaje,]) | Verifica que el objeto no sea nulo. |
| assertSame ([mensaje,] esperado, real) | Comprueba que ambas variables se refieren al mismo objeto. |
| assertNotSame ([mensaje,] esperado, real) | Comprueba que ambas variables se refieren a diferentes objetos. |

1- Utilizando Unit test

- ¿En el proyecto \*\*spring-boot\*\* para qué está esta dependencia en el pom.xml?

<dependency>

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

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

<scope>test</scope>

</dependency>

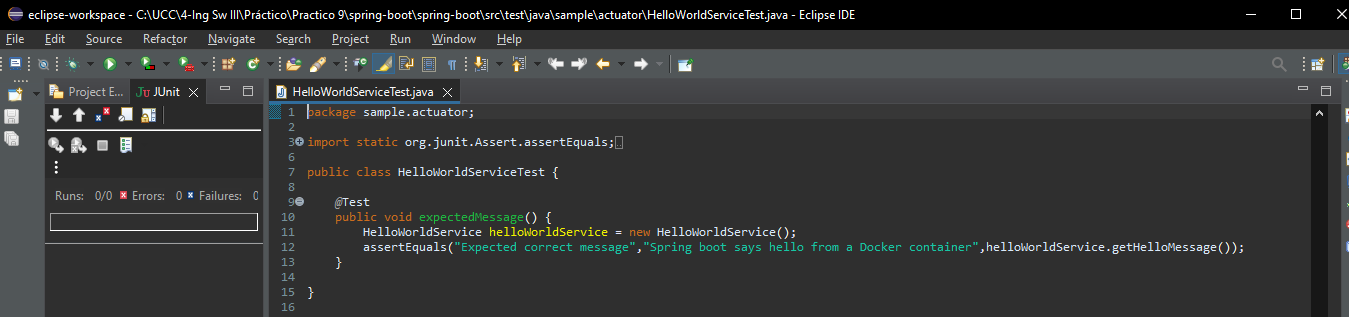
En archivo pom.xml del proyecto spring-boot contiene la dependencia spring-boot-starter-test, que es la principal contenida en la mayoría de los elementos necesarios para nuestras pruebas, es decir, proporciona dependencias de alcance de “pruebas” como JUnit, Hamcrest y Mockito con las siguientes bibliotecas:

JUnit: el estándar de facto para la prueba unitaria de app Java

Spring Test y Sprong Boot Test: utilidades y soporte de prueba de integración para app Spring boot

AssertJ: de aserciones fluída

- Analizar y ejectuar el metodo de unit test:



public class HelloWorldServiceTest {

@Test

public void expectedMessage() {

HelloWorldService helloWorldService = new HelloWorldService();

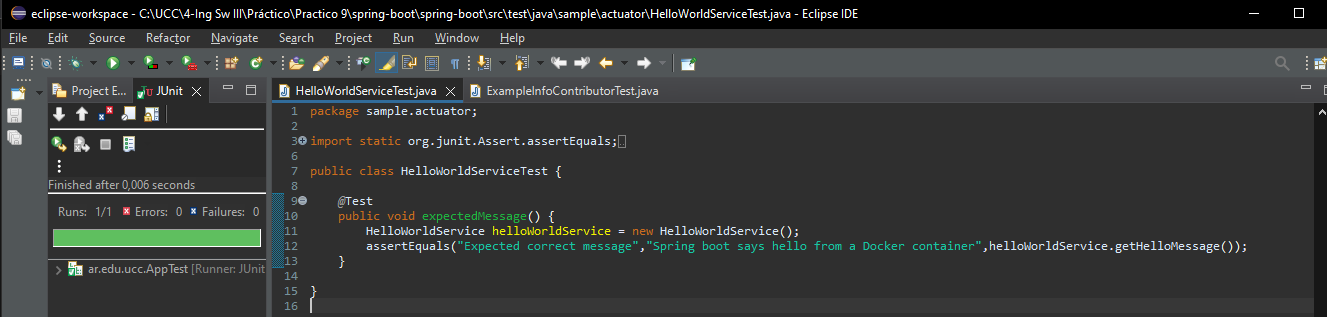
assertEquals("Expected correct message","Spring boot says hello from a Docker container",helloWorldService.getHelloMessage());

}

}

En la notación @Test del método HelloWorldServiceTest (de JUnit 4) identifica un método de prueba. Luego, se crea una instancia de la clase HelloWorldServiceTest y mediante la declaración AssertEquals, comprueba que el mensaje ingresado a través del método getHelloMessage() sea igual o retorne “Spring boot says hello from docker container”.

- Ejecutar los tests utilizando la IDE



**3- Familiarizarse con algunos conceptos de Mockito**

Mockito es un framework de simulación popular que se puede usar junto con JUnit. Mockito permite crear y configurar objetos falsos. El uso de Mockito simplifica significativamente el desarrollo de pruebas para clases con dependencias externas.

Si se usa Mockito en las pruebas, normalmente:

1. Se burlan las dependencias externas e insertan los mocks en el código bajo prueba

2. Se ejecuta el código bajo prueba

3. Se valida que el código se ejecutó correctamente

Referencia: https://www.vogella.com/tutorials/Mockito/article.html

- Analizar el código del test

public class ExampleInfoContributorTest {

@Test

public void infoMap() {

Info.Builder builder = mock(Info.Builder.class);

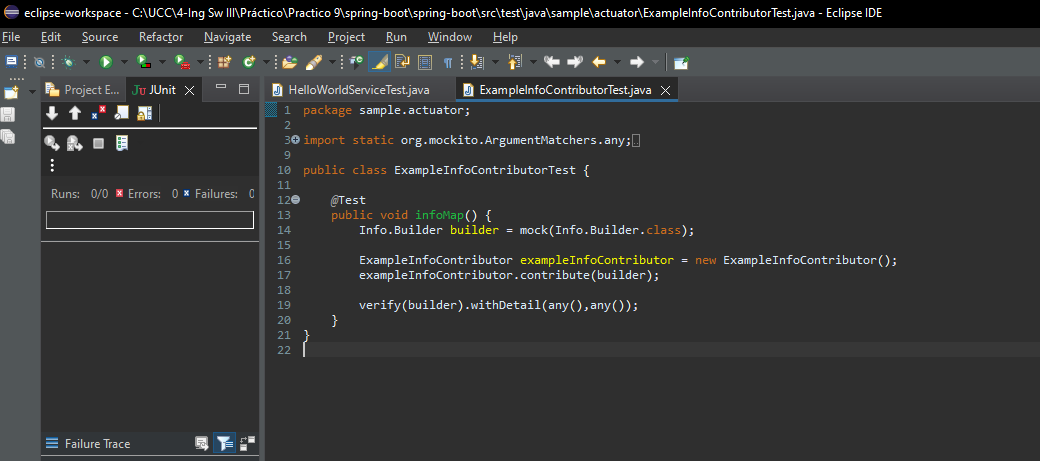
ExampleInfoContributor exampleInfoContributor = new ExampleInfoContributor();

exampleInfoContributor.contribute(builder);

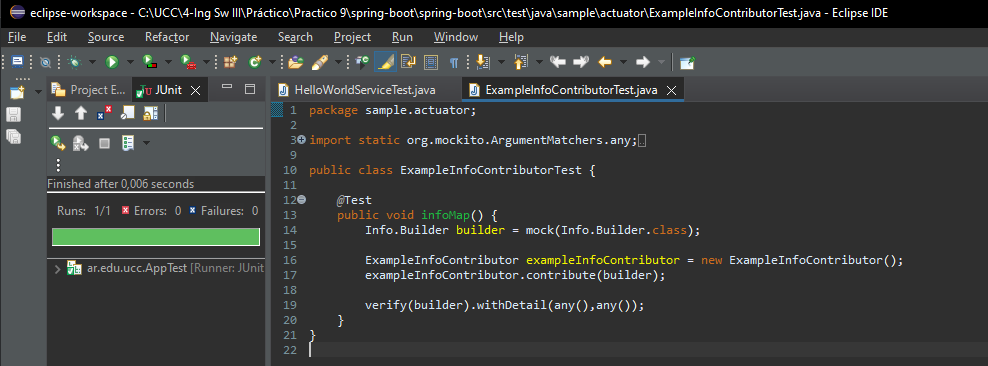
verify(builder).withDetail(any(),any());

}

}



La notación @Test (perteneciente a JUnit 4) identifica al método como un método de prueba. Se crea un objeto mock que es una implementación ficticia para una interfaz o una clase, en donde permite definir la salida de ciertas llamas a métodos, por lo general, registran la interacción con el sistema y las pruebas pueden validarlo, y esto le permite simplificar la configuración de la prueba. En este caso, el objeto mock llamado builder es una implementación ficticia para la clase Info.Builder.class. A continuación, se le asigna info con el método contribute() de la clase xampleInfoContributor. Al final, al ejecutarse, verifica que el mock contenga algún tipo de info o dato agregado.

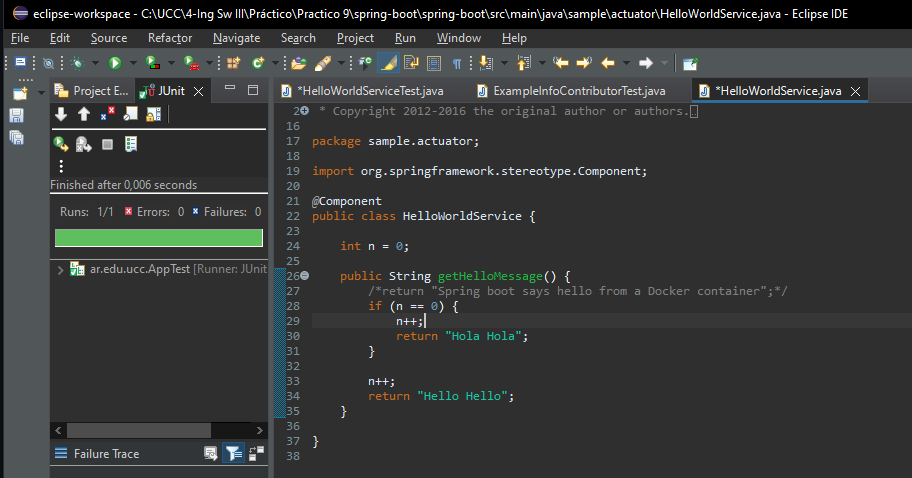


**4- Utilizando Mocks**

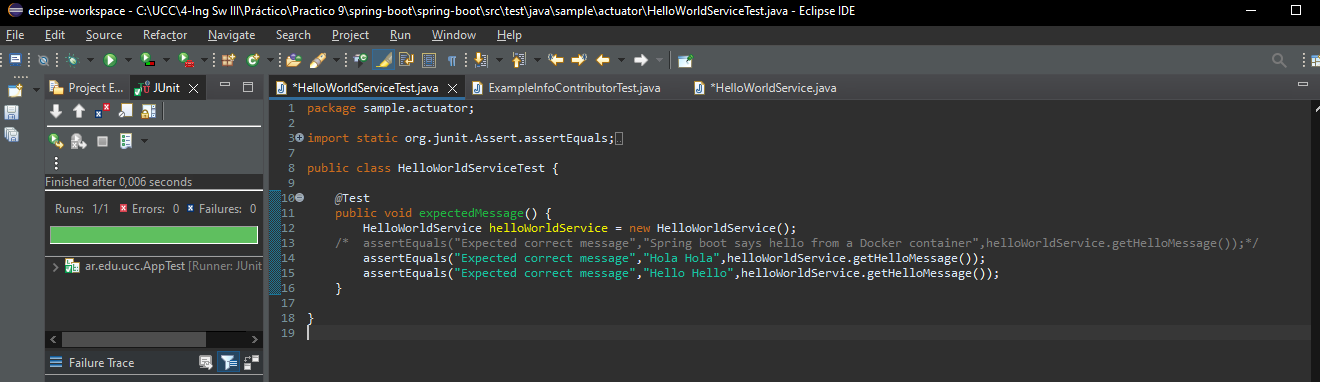
- Agregar un unit test a la clase \*\*HelloWorldServiceTest\*\*

- Cuando se llame por primera vez al método \*\*getHelloMessage\*\* retorne "Hola Hola"

- Cuando se llame por segunda vez al método \*\*getHelloMessage\*\* retorne "Hello Hello"



Primero, se tiene un contador inicializado en cero. Cuando se entra al método get, compara si el contador es igual a cero, entonces incrementa el contador porque se llamó al método por primera vez y retorna “Hola Hola”. La segunda llamada del método, teniendo ya el contador mayor que cero, se incrementa nuevamente y retorna “Hello Hello”.



El test es en donde se crea una instancia de la clase helloWorldService con la declaración assertEquals para que devuelva el mensaje de la primera llamada del método get de la instancia helloWorldService que sea igual a “Hola Hola”. Lo mismo ocurre con la segunda llamada al método mostrando el mensaje devuelto “Hello Hello”.

- Crear la siguiente clase \*\*AbstractTest\*\*

package sample.actuator;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import org.springframework.test.context.web.WebAppConfiguration;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringJUnit4ClassRunner.class)

@SpringBootTest(classes = SampleActuatorApplication.class)

@WebAppConfiguration

public abstract class AbstractTest {

protected MockMvc mvc;

@Autowired

WebApplicationContext webApplicationContext;

protected void setUp() {

mvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();

}

}

- Agregar esta otra clase también en el mismo directorio

package sample.actuator;

import static org.junit.Assert.assertEquals;

import org.junit.Before;

import org.junit.Test;

import org.springframework.http.MediaType;

import org.springframework.test.web.servlet.MvcResult;

import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

public class SampleControllerTest extends AbstractTest {

@Override

@Before

public void setUp() {

super.setUp();

}

@Test

public void testRootMessage() throws Exception {

String uri = "/";

MvcResult mvcResult = mvc.perform(MockMvcRequestBuilders.get(uri)

.accept( MediaType.APPLICATION\_JSON\_VALUE)).andReturn();

String content = mvcResult.getResponse().getContentAsString();

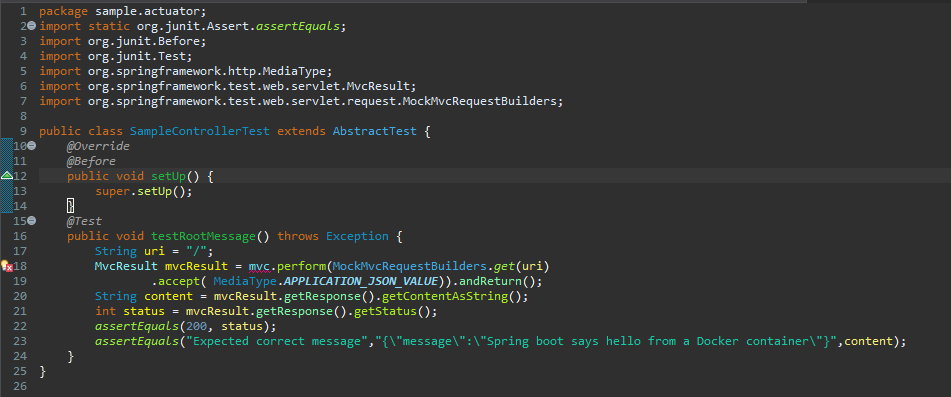
int status = mvcResult.getResponse().getStatus();

assertEquals(200, status);

assertEquals("Expected correct message","{\"message\":\"Spring boot says hello from a Docker container\"}",content);

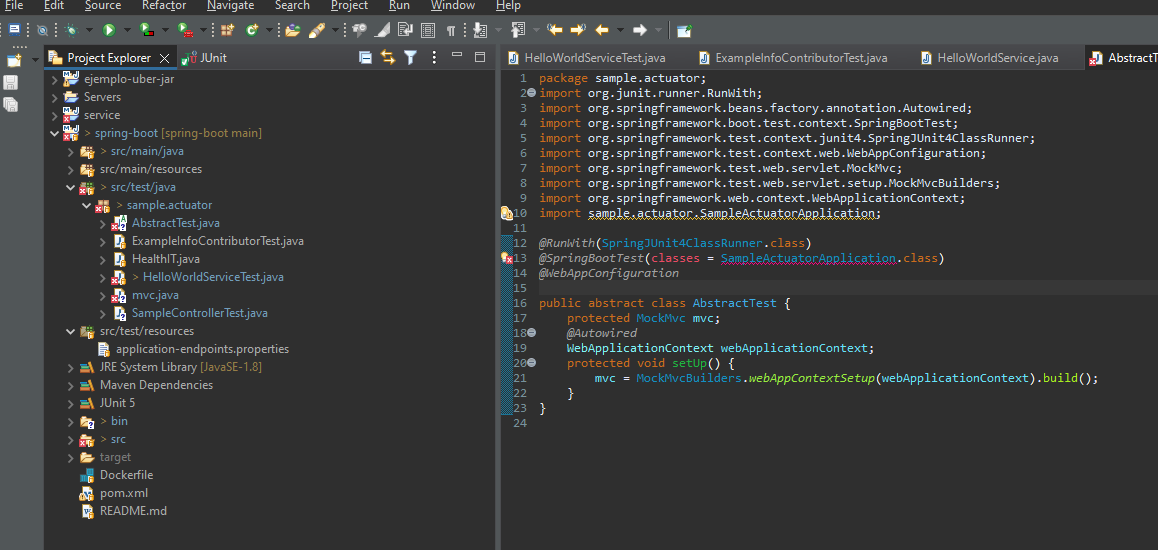
}

}

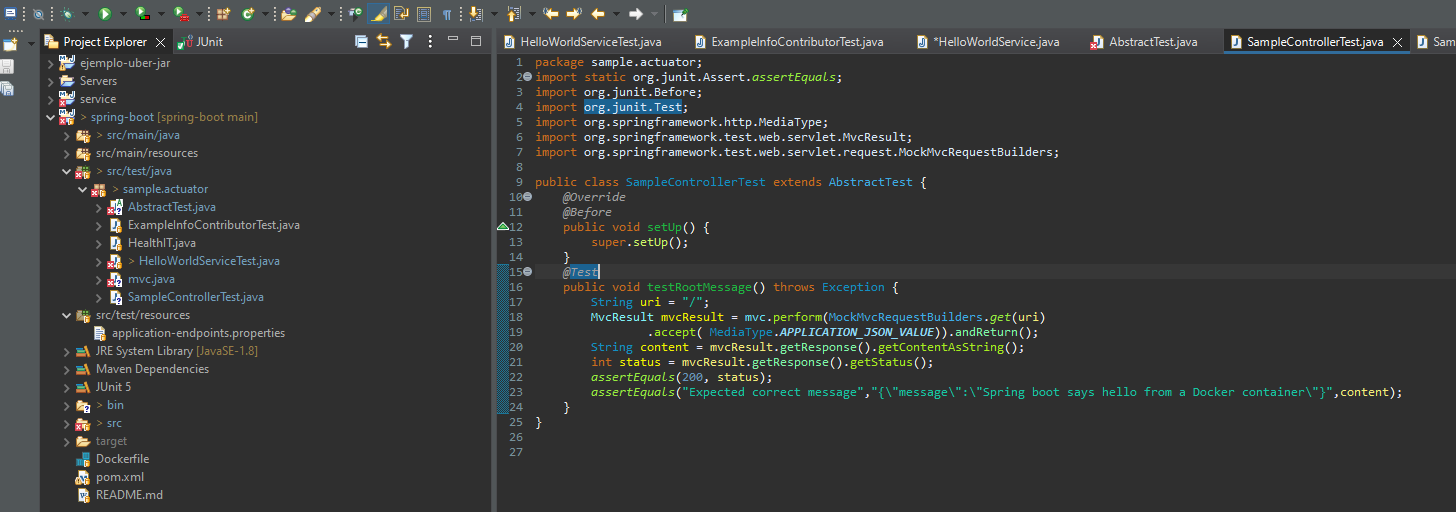


- Analizar estos tests

Se crear la clase tipo JUnit llamada AbstractTest. Aquí se instancia una variable mvc de tipo MockMVC, que define una “api fluida” para hacer una llamada web a través de una url con todos los parámetros que sean necesarios y valida que las respuestas sean correctas. La notación @Autowired usada por Spring Framework que permite inyectar dependencias con otras dentro Spring, y luego, crea el método setup que construye la variable mvc con un WebApplicationContext que se usa para crear la app web.



La clase JUnit SampleControllerTest hereda de AbstractTest, en donde recibe el método setUp que ejecuta antes del test que declara a continuación. Se construye un método TestRootMessage, que es un test por su notación @Test. A través de una petición o request http a la uri “/”, se guarda el repsonse en una variable mcvResult, verifica el status 200 que establece la conexión, devuelto por el response, y compara el contenido del repsonse con el contenido del archivo HelloWorldService.java



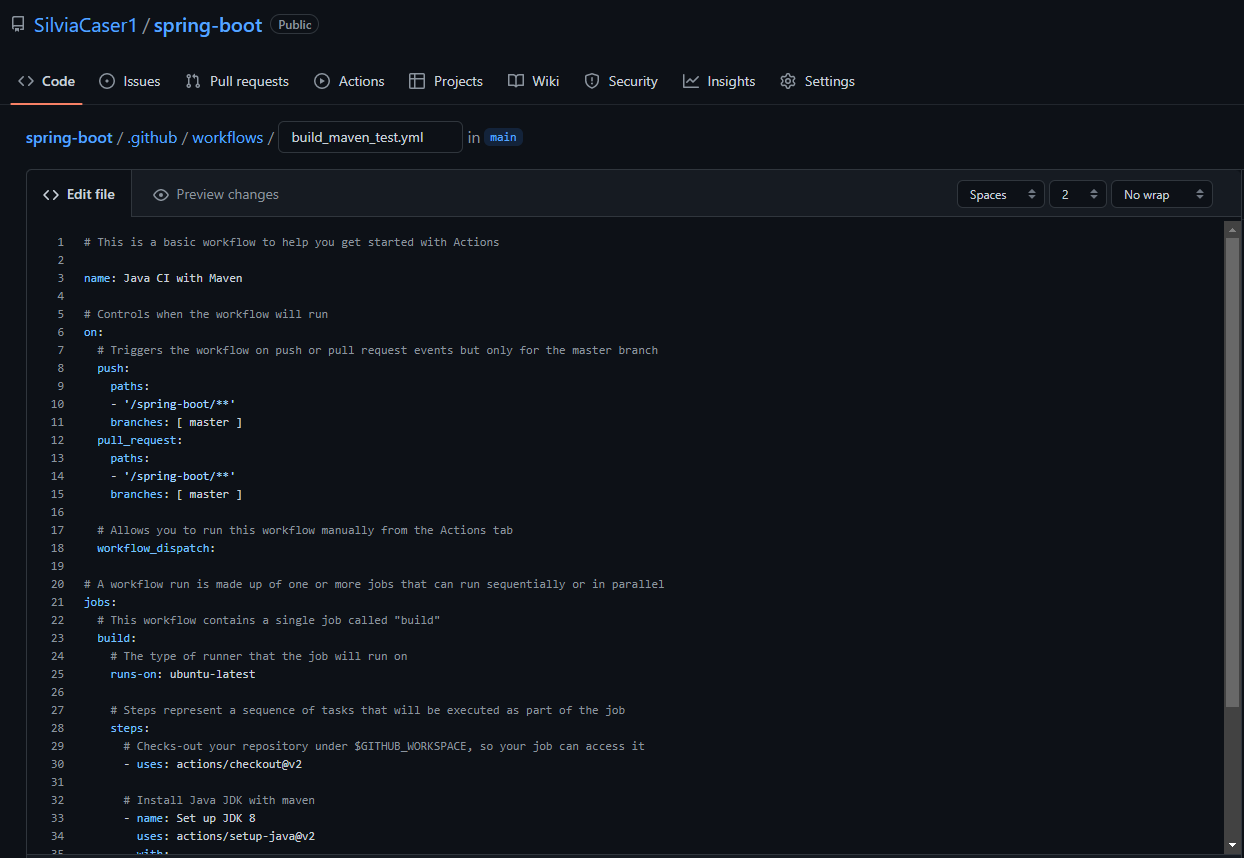
~~5- Opcional: Agregar otros unit tests~~

~~- Agregar unit tests para mejorar la cobertura, pueden ser test simples que validen getter y setters.~~

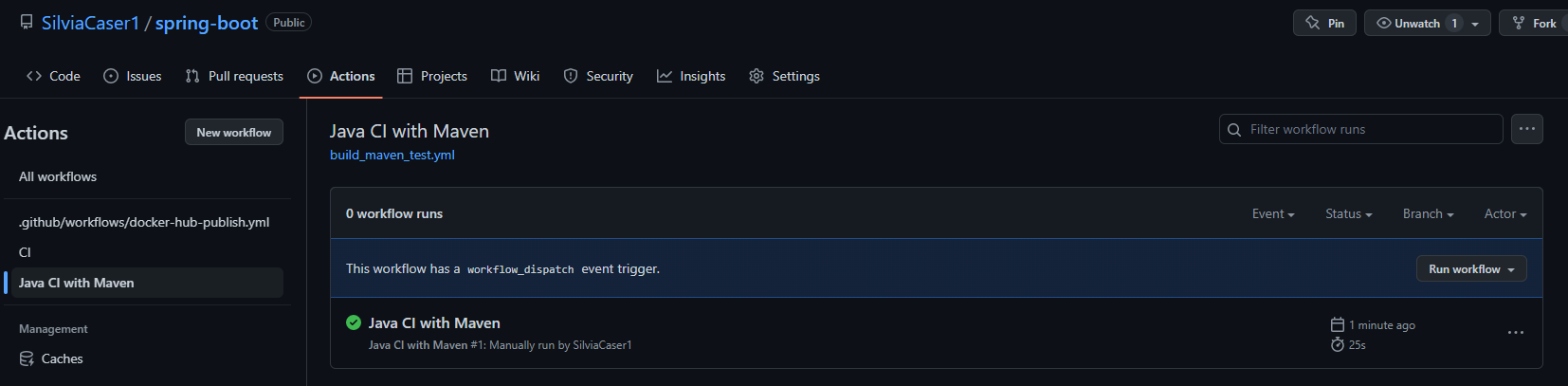
6- Capturar los unit tests como parte del proceso de CI/CD

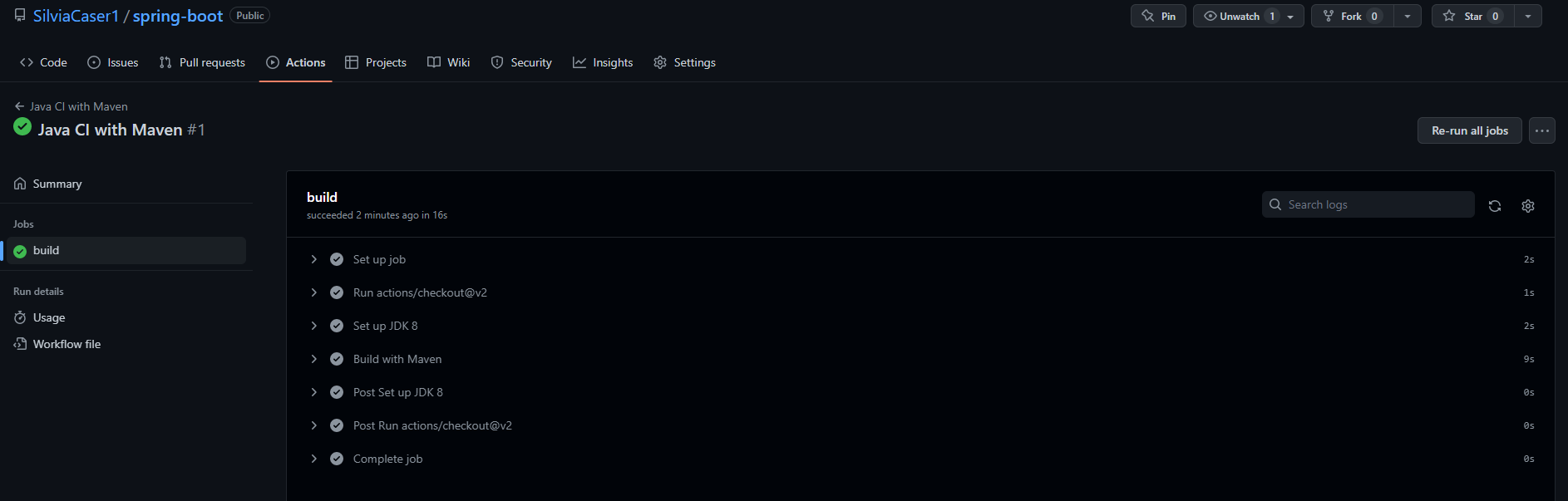
- Hacer los cambios en Jenkins (o en la herramienta de CICD utilizada) si es necesario, para capturar los resultados de los unit tests y mostrarlos en la ejecución del build.

Ir a GitHub Actions y crear el workflow build\_maven\_test.yml con el siguiente script:



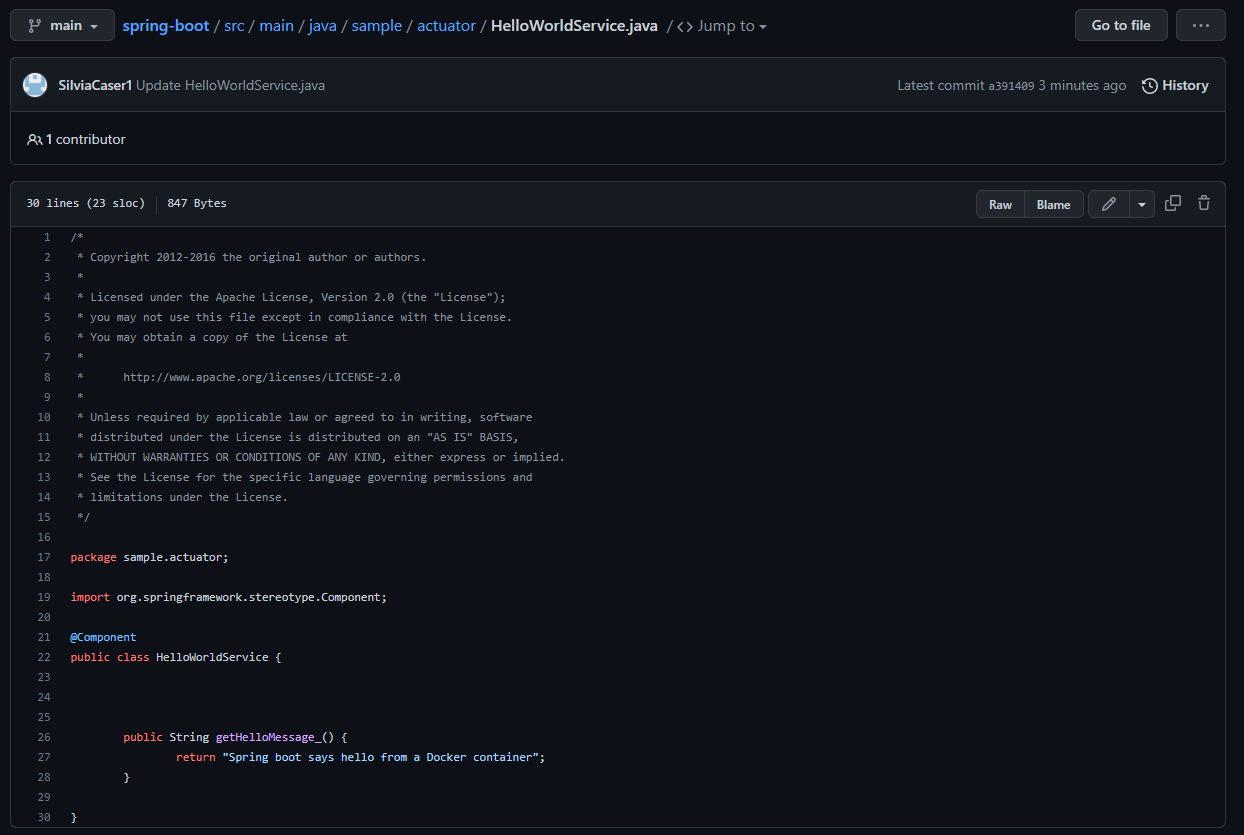
Commitear y correr manualmente



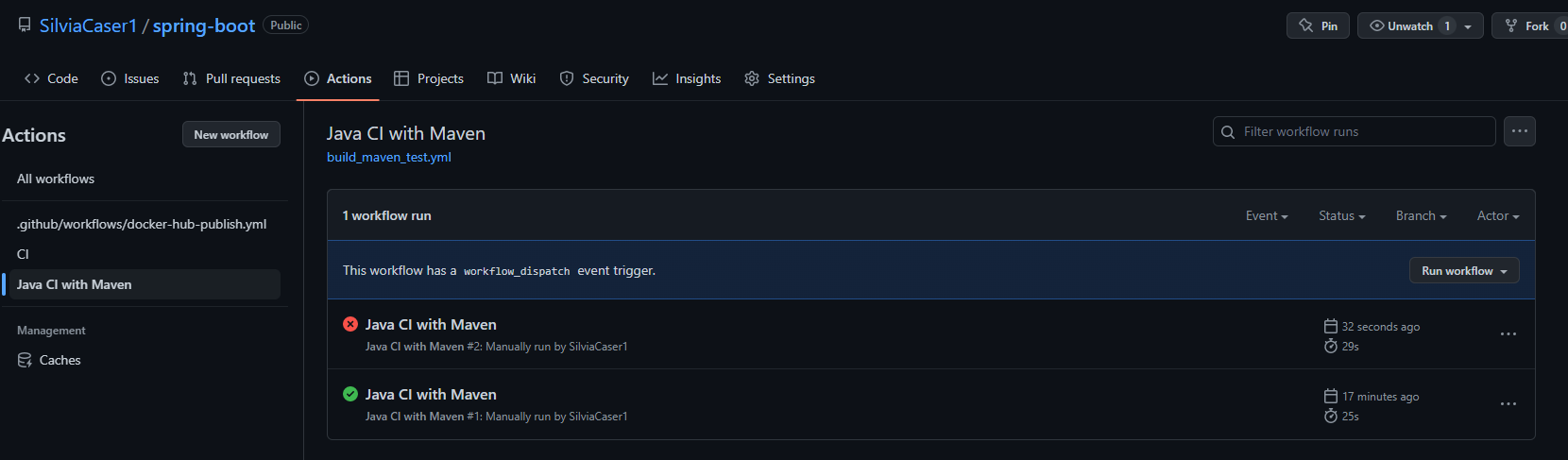


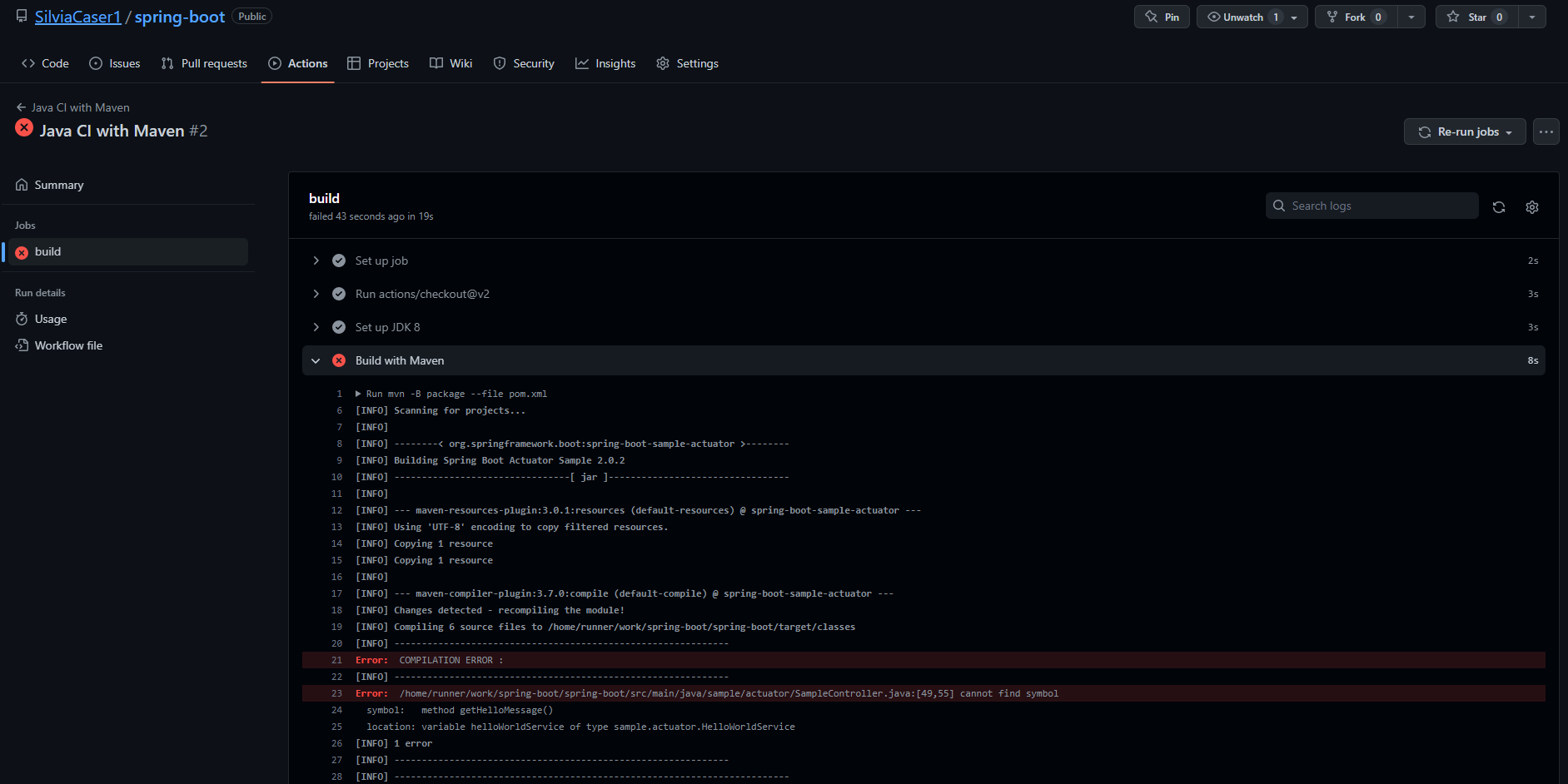
Modificar el archivo HelloWordlService.java para hacer que el test falle:

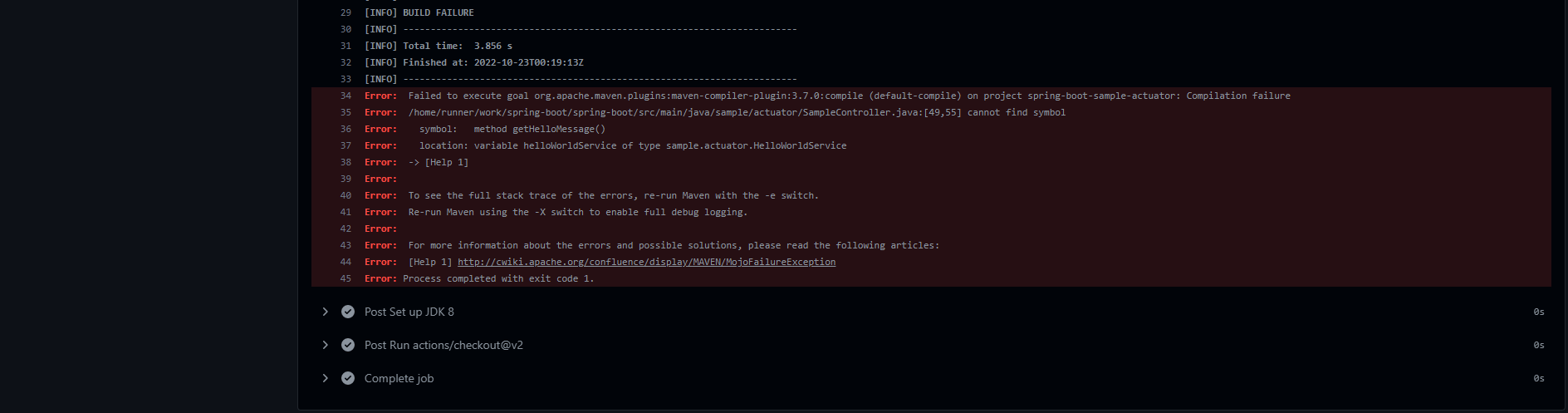
(modifiqué el nombre de la clase)



Corro el test







Corrijo y vuelvo a correr el test

