Procesos de la Ingeniería Software
Tema 4

Soporte Java para construcción de aplicaciones empresariales

6. Prueba de aplicaciones Jakarta EE

Prueba de aplicaciones Jakarta EE

- Pruebas Unitarias (Standalone)
  - Basadas en **JUnit** gracias a la naturaleza POJO de los componentes Jakarta EE
  - Utilizando objetos **Mock** para gestionar las dependencias
    - Tanto de otros componentes como de recursos del entorno (EntityManager, etc.)
- Pruebas de Integración
  - Se requiere la ejecución en un contenedor para verificar que funcionan sus servicios (inyección de dependencias, gestión del ciclo de vida, etc.)
  - Integración basada en prueba de métodos
    - Clases de prueba **JUnit**
    - Pruebas en **contenedor embebido**
      - Jakarta EE proporciona un contenedor embebido (**EJBContainer**) creado y lanzado desde la propia clase de prueba, es decir, desde una aplicación Java SE
    - Pruebas en **contenedor real**
      - Frameworks estilo **Arquillian**, que permiten lanzar el servidor real desde la propia clase de prueba
  - Integración basada en funcionalidad (end-to-end) / pruebas de interfaz
    - Sobre aplicación previamente desplegada en un servidor real
    - Usando frameworks estilo **Selenium** (aplicaciones web) o **FEST** (aplicaciones de escritorio)

Arquillian

- **Arquillian** es un framework para realizar pruebas de integración de aplicaciones Java/Jakarta EE
  - Independiente del servidor de aplicación

- Proporciona tres modos de ejecución de tests:
  - En contenedor embebido
    - Tests ejecutados sobre un contenedor embebido que se lanza desde la propia prueba (misma JVM)
  - En contenedor "managed"
    - Tests ejecutados sobre el contenedor (servidor) real pero su ciclo de vida lo gestiona Arquillian
  - En contenedor remoto
    - Tests ejecutados sobre el contenedor real y comunicación basada en protocolos estilo JMX

- Proporciona la capacidad de crear y desplegar archivos .jar, .war y .ear a través de la clase ShrinkWrap
  - Una vez desplegados podemos acceder a los beans a través de inyección de dependencias

Configuración Maven para uso de Arquillian con contenedor managed

- Definir una variable de entorno GLASSFISH_HOME apuntando al directorio de instalación de glassfish (al directorio "glassfish")

- Incluir en el fichero .pom las siguientes dependencias:
  - Arquillian para JUnit 5

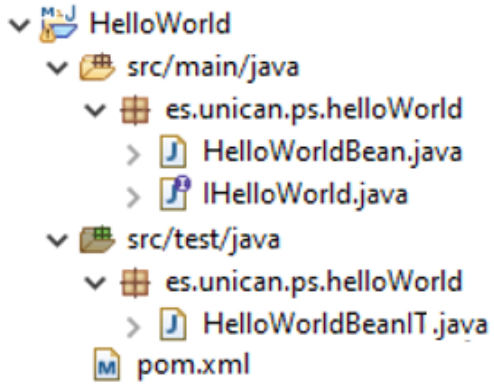
```
<dependency>
    <groupId>org.jboss.arquillian.junit5</groupId>
    <artifactId>arquillian-junit5-container</artifactId>
    <version>1.7.1.Final</version>
    <scope>test</scope>
</dependency>
```

  - Contenedor para Arquillian ("managed" y para Glassfish)

```
<dependency>
    <groupId>org.jboss.arquillian.container</groupId>
    <artifactId>arquillian-glassfish-managed-6</artifactId>
    <version>1.0.0.Alpha1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.glassfish.hk2</groupId>
    <artifactId>hk2</artifactId>
    <version>3.0.0-M2</version>
</dependency>
```

  - Añadir Maven-failsafe-plugin al apartado <build> para activar las pruebas de integración
    - https://maven.apache.org/surefire/maven-failsafe-plugin/usage.html
    - Las pruebas de activación se ejecutan con mvn verify y deben empezar o terminar por "IT"

Ejemplo básico de uso de Arquillian (con archivo .jar)




```java
@Remote
public interface IHelloWorld {
    public String hello(String name);
}
```

```java
@Stateless
public class HelloWorldBean implements IHelloWorld {
    public String hello(String name) {
        return "Hello "+name+"!";
    }
}
```

Test ejecutado con Arquillian

Bean inyectado

Generación del archivo de despliegue

Método ejecutado al lanzar la clase de prueba, antes de métodos @Test (despliega el archivo generado)

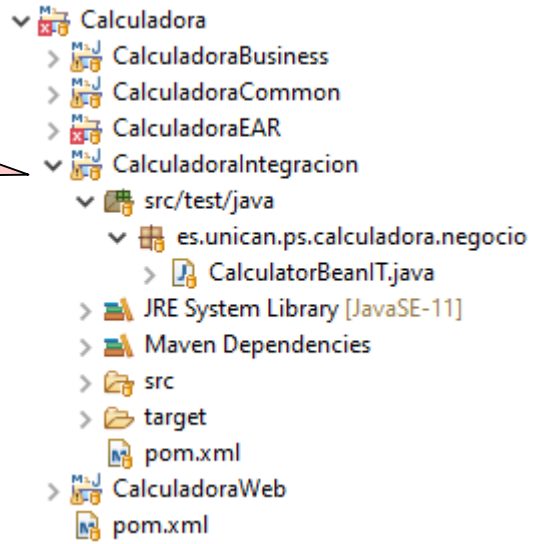
Test ejecutado sobre el bean inyectado una vez desplegado

```java
@ExtendWith(ArquillianExtension.class)
public class HelloWorldBeanIT {

    @EJB
    private IHelloWorld sut;

    @Deployment
    public static JavaArchive createDeployment() {
        JavaArchive jar = ShrinkWrap.create(JavaArchive.class);
        jar.addClass(HelloWorldBean.class);
        jar.addClass(IHelloWorld.class);
        return jar;
    }

    @Test
    public void testHello() {
       assertEquals(sut.hello("Patri"), "Hello Patri!");
    }
}
```

Ejemplo de Arquillian con archivos .ear

```java
@ExtendWith(ArquillianExtension.class)
public class CalculatorBeanIT {

    @EJB
    private ICalculatorRemote sut;

    @Deployment
    public static EnterpriseArchive createDeployment() {

        EnterpriseArchive ear = ShrinkWrap.createFromZipFile(EnterpriseArchive.class,
                                new File("../CalculadoraEAR/target/Calculadora.ear"));
        WebArchive war = ear.getAsType(WebArchive.class, "/es.unican.ps-CalculadoraWeb-0.0.1.war");
        war.addClass(CalculatorBeanIT.class);
        return ear;
    }

    @Test
    public void testAdd() {
        assertTrue(sut.add(3,5) == 8);
    }
}
```

Clases de prueba en un módulo Maven aparte (independencia de módulos)



Archivo .ear de despliegue

Hay que añadir la propia clase de test al modulo web o al modulo jar (porque en el ear en realidad no está)

Este ejemplo corresponde a la prueba del CalculatorBean usado en el tema 4.2 (proyectos disponibles en Moodle – Tema 4)

Ejecución de tests Arquillian

- Para una correcta ejecución de las pruebas con Arquillian:
  - Antes de ejecutar las pruebas
    - El servidor real (Glassfish en nuestro caso) debe estar apagado
    - El módulo que probamos no puede estar ya desplegado en el servidor
  - Después de ejecutar las pruebas
    - Puede ser necesario "limpiar" el servidor (para poder desplegar el módulo real)
      - Vaciar la carpeta /domains/domain1/applications
      - Eliminar posibles archivos .glassfishStaleFile de las carpetas correspondientes a los módulos probados