Jenkins

Herramientas de pruebas Java

Victor Herrero Cazurro

Contenidos

1. Pruebas de Software

1.1. Introducción

Son la parte del desarrollo, que persigue corroborar que el Software generado cumple con los requisitos planteados, son un extra en el desarrollo, no forman parte de la aplicación desarrollada.

Existen dos grandes grupos de pruebas

- Pruebas de caja blanca
- Pruebas de caja negra

Hay muchas formas de realizar las pruebas, algunas de ellas exigen la validacion visual de una persona que sepa que se esta probando y con que información, y que pueda estimar el resultado esperado, esta aproximación no es muy conveniente, dado que no permite automatizar el proceso de las pruebas.

Lo ideal será que la prueba contenga no solo el algoritmo de prueba, sino tambien las comprobaciones (asertos) del resultado, para ello existen en el mercado numerosos frameworks que permiten realizar una aproximación **Validación Rojo-Verde**.

1.1.1. Pruebas de caja blanca

Tipo de pruebas de software que se realiza sobre las funciones internas de un módulo, buscan recorrer todos los caminos posibles del modulo, cerciorándose de que no fallen.

Dado que probar todo es inviable en la mayor parte de los casos, se define **Cobertura** como una medida porcentual de cuánto código se ha cubierto.

NOTE	Las pruebas de caja blanca nos convencen de que un programa hace bien lo que hace, pero no de que haga lo que necesitamos.

1.1.2. Pruebas de caja negra

Se dice que una prueba es de caja negra cuando prescinde de los detalles del código y se limita a lo que se ve desde el exterior. Intenta descubrir casos y circunstancias en los que el módulo no hace lo que se espera de él, ejercitan los

requisitos funcionales.

NOTE	Las pruebas de caja negra están especialmente indicadas en aquellos módulos que van a ser interfaz con el usuario.

1.1.3. Cualidades deseables del Software

Existen un conjunto de cualidades, que se pueden evaluar de forma automatizada

- **Correcto**. Se comporta según los requisitos.
- **Robusto**. Se comporta de forma razonable, aun en situaciones inesperadas.
- **Confiable**. Se comporta según las expectativas del usuario (Correcto + Robusto).
- **Eficiente**. Emplea los recursos justos.
- **Verificable**. Sus características pueden ser comprobadas.

Y otras que no, que a lo maximo que se puede aspirar es a acotarlas, con arquitectura, buenas practicas, . . .

- **Amigable**. Fácil de utilizar.
- **Mantenible**. Se puede modificar fácilmente.
- **Reusable**. La misma pieza de software se puede usar sin cambios en otro proyecto.
- **Portable**. Es ejecutable en distintos ambientes (SO, . . .).
- **Legible**. El código es fácilmente interpretable.
- **Interoperable**. Puede interaccionar con otros software.

1.1.4. Most Important Test

- **Unitario**. Prueba un modulo lógico.
- **Integración**. Prueba un conjunto de módulos trabajando juntos buscando encontrar errores de forma temprana.
- **Regresión**. Determina si los cambios recientes en un módulo afectan a otros módulos.
- **Humo**. Pruebas de integración completa, ejecutadas de forma periódica, que buscan validar el funcionamineto basico de una aplicación antes de ser

entregada.

- **Sistema**. Verificación de que el ingreso, procesado y recuperación de datos se hace de forma correcta.
- **Aceptación**. Determinación por parte del cliente de si acepta el modulo.
- **Stress**. Comprobación del funcionamiento del sistema ante condiciones adversas, como memoria baja, gran cantidad de accesos y concurrencia en transacciones.
- **Carga**. Tiempo de respuesta para las transaciones del sistema, para diferentes supuestos de carga.

1.1.5. Most Important Metrics

- **Complejidad ciclomática**. Numero de caminos independientes en el código.
- **Cobertura**. Cantidad de código fuente cubierto por test.
- **Código duplicado**. Mide las veces que aparecen un numero de líneas repetidas (> 10 líneas).
- **Comentarios**. Cantidad de código que tiene documentación.
- **Diseño del software**. Indica el grado de acoplamiento de los módulos entre si.
- **Líneas**. Cantidad de líneas del código.
- **Malas practicas de codificación**. Aparición de numero mágicos, bloques de try sin procesar, . . .

1.2. Pruebas Unitarias

Una prueba unitaria es una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

En las pruebas unitarias se prueban clases concretas, no conjuntos de clases, es decir una prueba unitaria prueba la funcionalidad de un método de una clase suponiendo que los objetos que emplea realizan sus tareas de forma correcta.

1.3. JUnit

Framework para pruebas.

Paquetes junit.* (JUnit 3) y org.junit.* (JUnit 4) Embedido en Eclipse (JUnit 3 y 4),

eclipse proporciona la creación de Junit Test Case (caso de prueba) y Junit Test Suite (conjunto de casos de pueba).

JUnit 4 admite timeout, excepciones esperadas, tests ignorables, test parametrizados, . . .

1.3.1. Test Suites

Conjunto de pruebas a ejecutar de forma conjunta.

```
@RunWith(Suite.class)
@SuiteClasses({C1Test.class, C2Test.class})
public class TestSuite{

}
```

1.3.2. Ciclo de vida de los Test

Se proporcionan anotaciones que permiten actuar en las distintas fases del ciclo de vida

- @Before: El método de instancia anotado con esta anotación, se ejecutara antes de cada Test de la clase, por tanto tantas veces como métodos de instancia anotados con @Test existan.
- @After: El método de instancia anotado con esta anotación, se ejecutara despues de cada Test de la clase, por tanto tantas veces como métodos de instancia anotados con @Test existan.
- @BeforeClass: El método estatico anotado con esta anotación, se ejecutara antes de cualquier otro de la clase y solo una vez.
- @AfterClass: El método estatico anotado con esta anotación, se ejecutara despues de todos los otros métodos de instancia de la clase y solo una vez.
- @Test: El método anotado con esta anotación, representa un Test.
- @Ignore: Permite ignorar un método de Test.

1.3.3. Probando el lanzamiento de excepciones

La anotacion @Test, permite realizar pruebas, donde el resultado esperado sea una **Excepcion**.

```
@Test(expected=InvalidIngresoException.class)
public void comprobamosQueLanzamosException() throws
InvalidIngresoException{
}
```

1.3.4. Probando restricciones temporales

La anotacion @Test, permite realizar pruebas, donde el tiempo transcurrido en la ejecución esta limitado por el requisito.

```
@Test(timeout=12000)
public void testDeRendimiento() {
}
```

1.3.5. Test parametrizados

El API incorpora un **Runner**, que permite la ejecución repetida de Test, cada vez con unos datos de prueba, para lo cual hay que

- Anotar la clase con **@RunWith**

```
@RunWith(Parameterized.class)
```

- Crear un método publico estatico que retorne un **Array de Arrays**, anotado con **@Parameters**

```
@Parameters
public static Collection<Object[]> data() {}
```

NOTE

Puede ser interesante emplear la clase de utilidad **Arrays**

```
Arrays.asList(new Object[][] {{},{},{}});
```

- Crear Atributos de clase de la misma tipologia que los elementos de los Arrays.
- Constructor que establezca los atributos.

1.3.6. Asertos

Los asertos, son métodos estaticos del API, que permiten realizar validaciones, para poder comprobar que los datos obtenidos estan dentro del rango esperado.

Algunos de los asertos que se proporcionan son:

- assertEquals
- assertFalse
- assertTrue
- assertNotNull
- assertNull
- assertNotSame
- assertSame
- fail

NOTE

Implementar una clase que obtenega los impuestos según los ingresos siguiendo las siguientes reglas:

- Ingreso < = 8000 no paga impuestos.
- 8000 < Ingreso < = 15000 paga 8% de impuestos.
- 15000 < Ingreso < = 20000 paga 10% de impuestos.
- 20000 < Ingreso < = 25000 paga 15% de impuestos.
- 25000 < Ingreso paga 19.5% de impuestos.

Creamos una clase con un método calcularImpuestosPorIngresos, que recibiendo una cantidad double como parámetro, que son los ingresos de una persona, retornará los impuestos que ha de pagar dicha persona (double).

1.4. Hamcrest

Framework especializado en proporcionar asertos mas semanticos, permite realizar los Test, con un lenguaje mas cercano, mas legible.

La librería hamcrest-library, proporciona una clase con métodos estáticos **org.hamcrest.Matchers**.

Estos métodos estáticos, se emplean en formar un predicado, que forma el aserto, para que la forma de leer el código sea mas natural.

Algunos de los métodos estáticos de Hamcrest.

- is
- not
- nullValue
- empty
- endsWith
- startsWith
- hasItem
- hasItems
- hasProperty

Un ejemplo de predicado con **Hamcrest**, podria ser el siguiente, donde se **valida que** un objeto **calculadora es no nulo**. La lectura comprensiva de la sentencia, coincide con los escrito.

```
assertThat(calculadora, is(not(nullValue())));
```

Otro ejemplo, que **valida que** el objeto **persona tiene la propiedad** "nombre".

```
assertThat(persona, hasProperty("nombre"));
```

1.5. Mockito

Para poder crear un buen conjunto de pruebas unitarias, es necesario centrarse exclusivamente en la unidad (normalmente será un metodo de una clase concreto) a testear, para ello se pueden simular, con **Mocks** el resto de clases involucradas, de esta manera se crean test unitarios potentes que permiten detectar los errores allí donde se producen y no en dependencias del supuesto código probado.

Mockito es una herramienta que permite generar **Mocks** dinámicos. Estos pueden ser de clases concretas o de interfaces. Esta parte de la generación de las pruebas, no se centra en la validación de los resultados, sino en los que han de retornar

aquellos componentes de los que depende la clase probada.

La creación de pruebas con Mockito se divide en tres fases

- **Stubbing**: Definición del comportamiento de los Mock ante unos datos concretos.
- **Invocación**: Utilización de los Mock, al interaccionar la clase que se esta probando con ellos.
- **Validación**: Validación del uso de los Mock.

Se pueden definir los **Mock** con

- La anotacion @Mock aplicada sobre un atributo de clase.

```
@Mock
private IUserDAO mockUserDao;
```

De emplearse las anotaciones, se ha de ejecutar la siguiente sentencia para que se procesen dichas anotaciones y se generen los objetos **Mock**

```
MockitoAnnotations.initMocks(testClass);
```

O bien emplear un **Runner** especifico de Mockito en la clase de Test que emplee Mockito, el **MockitoJUnitRunner**

```
@RunWith(MockitoJUnitRunner.class)
```

- O con el método estatico **mock**.

```
private IDataSesionUserDAO mockDataSesionUserDao = mock
(IDataSesionUserDAO.class);
```

1.5.1. Stubing

Se persigue definir comportamientos del **Mock**, para ello se emplean los métodos estaticos de la clase **org.mockito.Mockito**, que son

- atLeast

- atMost
- atLeastOnce
- doNothing
- doReturn
- doThrow
- when
- inOrder
- never
- only
- verify
- mock

Y de la clase **org.mockito.Matchers**, que son

- any
- anyString
- anyObject
- contains
- endsWith
- startsWith
- eq
- isA
- isNull
- isNotNull

Algunos ejemplos de definicion de comportamientos del Mock

```
when(mockUserDao.getUser(validUser.getId())).thenReturn(validUser);

when(mockUserDao.getUser(invalidUser.getId())).thenReturn(null);

when(mockDataSesionUserDao.deleteDataSesion((User) eq(null), anyString
())).thenThrow(new OperationNotSupportedException());

when(mockDataSesionUserDao.updateDataSesion(eq(validUser), eq(validId),
anyObject())).thenReturn(true);

when(mockDataSesionUserDao.updateDataSesion(eq(validUser), eq(
invalidId), anyObject())).thenThrow(new OperationNotSupportedException
());

when(mockDataSesionUserDao.updateDataSesion((User) eq(null), anyString
(), anyObject())).thenThrow(new OperationNotSupportedException());
```

Por defecto todos los métodos que devuelven valores de un mock devuelven null, una colección vacía o el tipo de dato primitivo apropiado, salvo que se defina un comportamiento distinto.

1.5.2. Verificación

Se puede verificar el orden en el que se han ejecutado los métodos del **Mock**, pudiendo llegar a diferenciar el orden de invocación de un mismo método por los parametros enviados.

En este ejemplo se esta verificando que el orden de ejecucion de los métodos **getUser** *del mock* **mockUserDao**, *se ejecuta antes que el método* **deleteDataSesion** *del mock* **mockDataSesionUserDao**

```
ordered = inOrder(mockUserDao, mockDataSesionUserDao);
ordered.verify(mockUserDao).getUser(validUser.getId());
ordered.verify(mockDataSesionUserDao).deleteDataSesion(validUser,
validId);
```

Tambien se puede verificar el numero de veces que se ha invocado una funcionalidad

En este ejemplo se verifica que el método **someMethod** *del* **mock** *no se ejecuta nunca, que el método* **someMethod(int)** *se ejecuta 1 sola vez y que el método* **someMethod(string)** *se ejecuta 2 veces.*

```
verify(mock, never()).someMethod();
verify(mock, only()).someMethod(2);
verify(mock, times(2)).someMethod("some arg");
```

1.6. Selenium

Paquete de herramientas para automatizar pruebas de aplicaciones Web en distintas plataformas.

La documentación la podremos obtener aquí

Las herramientas que componen el paquete son

- Selenium IDE.
- Selenium Remote Control (RC) o selenium 1.
- Selenium WebDriver o selenium 2.

1.6.1. Selenium IDE

Se trata de un plugin de Firefox, que nos permitirá grabar y reproducir una macro con una prueba funcional, la cual podremos repetir las veces que deseemos. Se puede descargar aquí

Las acciones que se realizan en la navegación mientras se graba la macro, se traducen en comandos.

La macro por defecto se guarda en HTML, aunque también se puede obtener como java, c#, Python, . . .

También se podrán insertar validaciones y no solo acciones sobre la pagina, aunque las validaciones son mas faciles de escribir en el código generado (java, c#, . . .)

Una vez grabada la macro, el HTML que se genera tiene una tabla con 3 columnas:

- Comando de Selenium.
- Primer parámetro requerido

- Segundo parámetro opcional

Los comandos de selenium se dividen en tres tipos:

- Acciones– Acciones sobre el navegador.
- Almacenamiento– Almacenamiento en variables de valores intermedios.
- Aserciones– Verificaciones del estado esperado del navegador.

Los comandos de navegación mas habituales son:

- **open**: abre una página empleando la URL.
- **click/clickAndWait**: simula la acción de click, y opcionalmente espera a que una nueva pagina se cargue.
- **waitForPageToLoad**: para la ejecución hasta que la pagina esperada es cargada. Es llamada por defecto automáticamente cuando se invoca clickAndWait.
- **waitForElementPresent**: para la ejecución hasta que el UIElement esperado, esta definido por un tag HTML presente en la pagina.
- **chooseCancelOnNextConfirmation**: Predispone a seleccionar en la próxima ventana de confirmación el botón de Cancel.

Los comandos de almacenamiento mas habituales son:

- **store**: Almacena en la variable el valor.
- **storeElementPresent**: Almacenara True o False, dependiendo de si encuentra el UI Element.
- **storeText**: Almacena el texto encontrado. Es usado para localizar un texto en un lugar de la pagina especifico.

Los comandos de verificación mas habituales son:

- **verifyTitle/assertTitle**: verifica que el titulo de la pagina es el esperado.
- **verifyTextPresent**: verifica que el texto esperado esta en alguna parte de la pagina.
- **verifyElementPresent**: verifica que un UI element esperado, esta definido como tag HTML en la presente pagina.
- **verifyText**: verifica si el texto esperado y su tag HTML estan presentes en la pagina.

- **assertAlert**: verifica si sale un alert con el texto esperado.
- **assertConfirmation**: verifica si sale una ventana de confirmacion con el texto esperado.

1.6.2. Selenium WebDriver

Es el motor de pruebas automatizadas de Selenium, se encarga de arrancar un navegador que responde a las ordenes del Test, provocando la ejecución de la macro grabada.

No todas las versiones de Firefox son compatibles con **Selenium WebDriver**, se puede encontrar mas información aquí o aquí

Para descargar versiones antiguas de Firefox, se puede hacer desde aquí

NOTE	Se puede probar con la version de selenium 2.52.0 y Firefox 45.

Para la dependencia del proyecto con selenium webdriver, añadir

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>2.19.0</version>
</dependency>
```

El codigo obtenido de la macro grabada con Selenium IDE, tendra las siguientes sentencias

- La creación del objeto que representa la interaccion con el navegador

```
FirefoxDriver driver = new FirefoxDriver();
```

- La peticion

```
driver.get(baseUrl + "/05-Servidor/");
```

2. Cobertura

2.1. Cobertura

La Cobertura, representa la cantidad de codigo que cubren las pruebas realizadas sobre el codigo.

Existe un plugin de Maven que permite realizar la medición de la cobertura, presentando el resultado en informes **html** y **xml**.

```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>cobertura-maven-plugin</artifactId>
            <version>2.5.2</version>
            <configuration>
                <formats>
                    <format>xml</format>
                    <format>html</format>
                </formats>
            </configuration>
        </plugin>
    </plugins>
</reporting>
```

2.2. Jacoco

Formado por las primeras silabas de **Java Code Coverage**, es otro plugin de cobertura.

La página de referencia se encuentra aquí

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.7.5.201505241946</version>
            <executions>
                <execution>
```

```
                    <id>pre-unit-test</id>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                    <configuration>
                        <!-- Establece la ubicacion del fichero con los
datos de la ejecucion. -->
                        <destFile>${project.build.directory}/jacoco-
ut.exec</destFile>
                        <!-- Establece la propiedad que contiene la
ruta del agente Jacoco para las pruebas unitarias-->
                        <propertyName>surefireArgLine</propertyName>
                    </configuration>
                </execution>
                <execution>
                    <id>post-unit-test</id>
                    <phase>test</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                    <configuration>
                        <!-- Establece la ubicacion del fichero con los
datos de la ejecucion. -->
                        <dataFile>${project.build.directory}/jacoco-
ut.exec</dataFile>
                        <!-- Establece la ruta donde se genera el
reporte para pruebas unitarias -->

<outputDirectory>${project.reporting.outputDirectory}/jacoco-
ut</outputDirectory>
                    </configuration>
                </execution>
                <execution>
                    <id>pre-integration-test</id>
                    <phase>pre-integration-test</phase>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                    <configuration>
                        <!-- Establece la ubicacion del fichero con los
datos de la ejecucion. -->
                        <destFile>${project.build.directory}/jacoco-
it.exec</destFile>
                        <!-- Establece la propiedad que contiene la
ruta del agente Jacoco para las pruebas de integracion -->
```

```
                        <propertyName>failsafeArgLine</propertyName>
                    </configuration>
                </execution>
                <execution>
                    <id>post-integration-test</id>
                    <phase>post-integration-test</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                    <configuration>
                        <!-- Establece la ubicacion del fichero con los
datos de la ejecucion. -->
                        <dataFile>${project.build.directory}/jacoco-
it.exec</dataFile>
                        <!-- Establece la ruta donde se genera el
reporte para pruebas de integracion -->

<outputDirectory>${project.reporting.outputDirectory}/jacoco-
it</outputDirectory>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Se ha de configurar igualmente que el agente sea ejecutado en las distintas fases de **test** e **integration-test**, indicando en el plugin con **argLine** la ubicacion del agente de **jacoco** que debera generar los datos del analisis.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.12.4</version>
            <configuration>
                <argLine>${surefireArgLine}</argLine>
                <excludes>
                    <exclude>**/integracion/*.java</exclude>
                </excludes>
                <includes>
                    <include>**/unitarias/*.java</include>
                </includes>
```

```
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>2.8</version>
            <configuration>
                <argLine>${failsafeArgLine}</argLine>
                <excludes>
                    <exclude>**/unitarias/*.java</exclude>
                </excludes>
                <includes>
                    <include>**/integracion/*.java</include>
                </includes>
            </configuration>
            <executions>
                <execution>
                    <id>pasar test integracion</id>
                    <phase>integration-test</phase>
                    <goals>
                        <goal>integration-test</goal>
                    </goals>
                </execution>
                <execution>
                    <id>validar pruebas integracion</id>
                    <phase>verify</phase>
                    <goals>
                        <goal>verify</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Finalmente se pueden añadir los resultados del analisis al sitio añadiendo

```
<reporting>
    </plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.7.5.201505241946</version>
        </plugin>
    </plugins>
</reporting>
```

3. SOA

La Arquitectura Orientada a Servicios (SOA), es una arquitectura para diseñar sistemas distribuidos, buscando facilidad y flexibilidad de integración de los componentes que lo forman, siendo sistemas altamente escalables.

Define la utilización de componentes (servicios) para implementar la lógica de negocio.

3.1. Conceptos

Servicio Funcionalidad sin estado, auto-contenida, que acepta llamadas y devuelve respuestas mediante una interfaz bien definida.

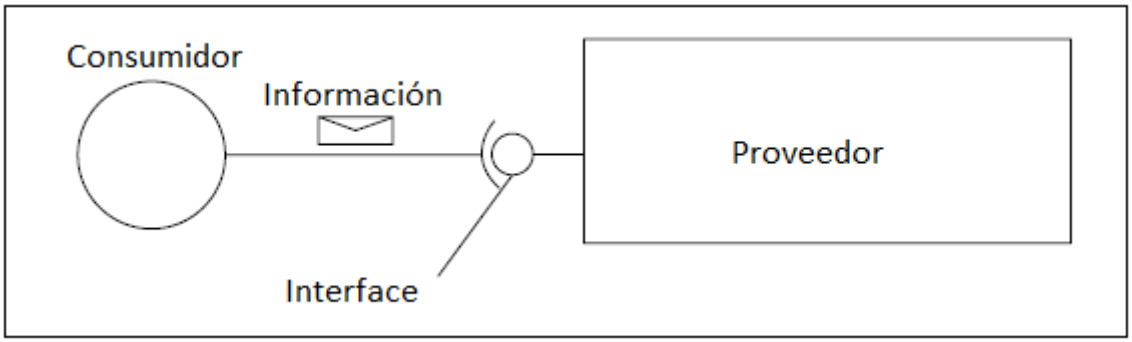
Orquestación Secuenciación de la ejecución de los servicios necesarios para responder a los requisitos. No incluye la presentación de los datos.

Sin estado Los servicios no mantienen ni dependen de condiciones pre-existente.

Proveedor Cada componente que forma la arquitectura será proveedor de una funcionalidad y quizas tambien a su vez sea Consumidor de otra.

Consumidor Cada componente que invoca la funcionalidad de otro componente, el principal será el orquestador.

Interface Contrato establecido entre el Proveedor y el Consumidor, para conseguir un perfecto entendimiento.



3.2. Estandares

SOA para conseguir la interoperabilidad, promueve basar los desarrollos en estandares, aunque si bien estos no son obligatorios, son los mas habituales y recomendados

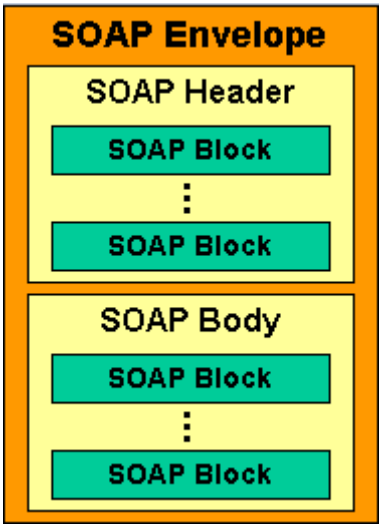
- XML
- HTTP
- SOAP
- WSDL
- UDDI
- JSON

4. SOAP

El Simple Object Access Protocol (SOAP), es un protocolo que establece como se invoca a un servicio remoto.

Emplean XML para la representación de la información. Los mensajes XML se estructuran en

- Envelope
  - ¬ Header
  - ¬ Body



Emplean HTTP, SMTP, TCP o JMS como protocolo de transporte.

Es el protocolo mas empleado en arquitecturas SOA.

5. SoapUI

Herramienta que permite crear facilmente pruebas funcionales (unitarias y integracion) y no funcionales (carga y rendimiento) de servicios web.

5.1. Caracteristicas

- Compatibilidad con la mayoria de estandares relacionados con los servicios web.
- Posiblidad de crear Mocks de servicios.
- Permite crear pruebas funcionales.
- Permite crear pruebas de rendimiento.
- Se puede integrar con herramientas como Maven o servidores de IC como Jenkins.

5.2. Instalación

Para la instalación, descargar de aquí.

El producto tiene dos versiones

- OpenSource
- Comercial

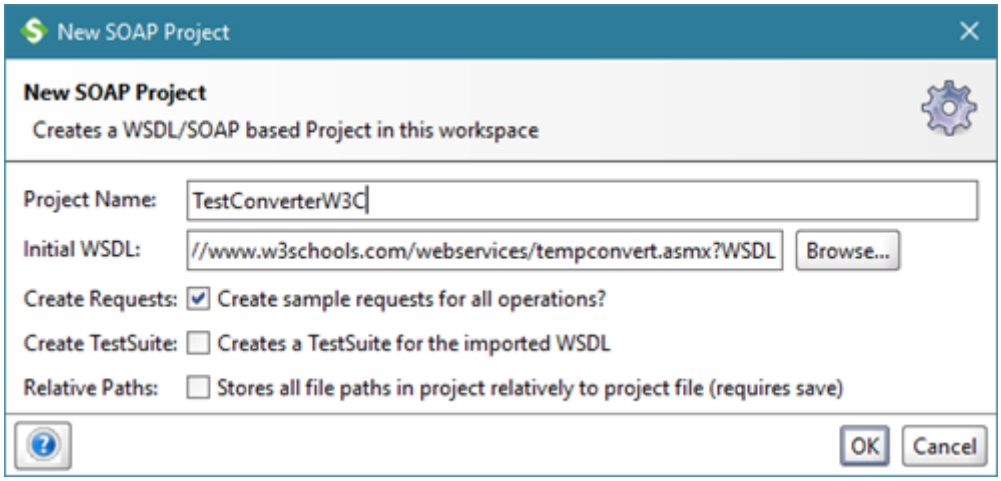
5.3. Pruebas de servicios web SOAP

Se pueden encontrar servicios web publicos para realizar pruebas aquí.

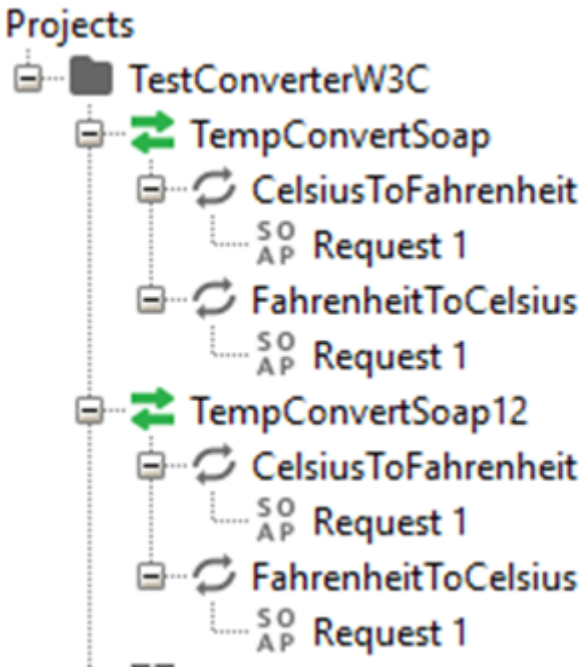
5.3.1. Procedimiento

Crear un nuevo proyecto SOAP, indicando

- Nombre
- WSDL
- Marcar "Crear ejemplos de peticiones para todas las operaciones"



La herramienta creará el proyecto con * Una entrada por cada Port. * Una entrada por cada Operation de cada Port-Binding. * Una Request por cada Operation.

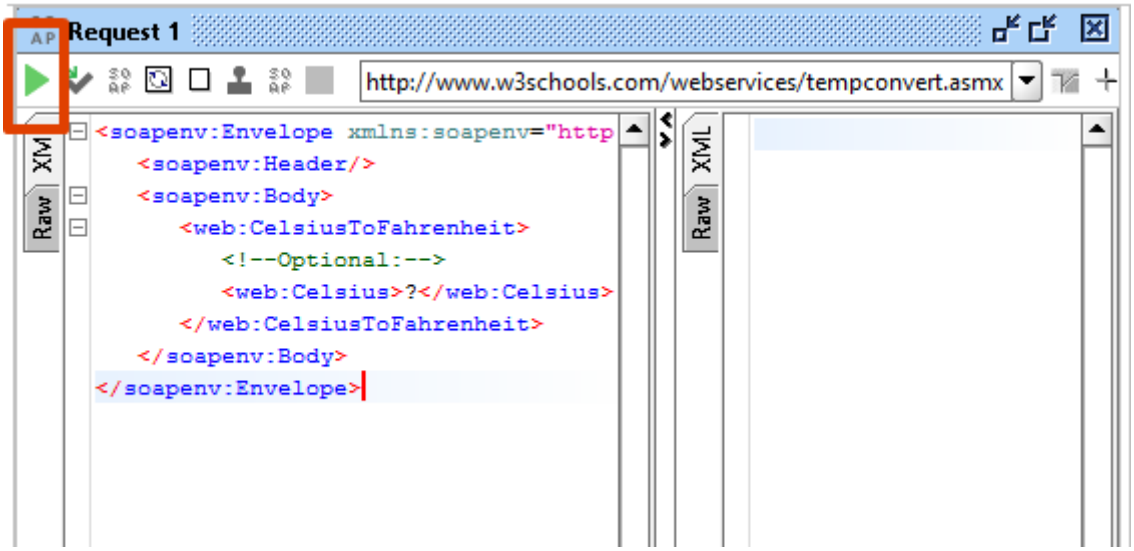


Sobre el Port generado, se puede pinchar y se accede a una ventana que permite navegar por las características del Port

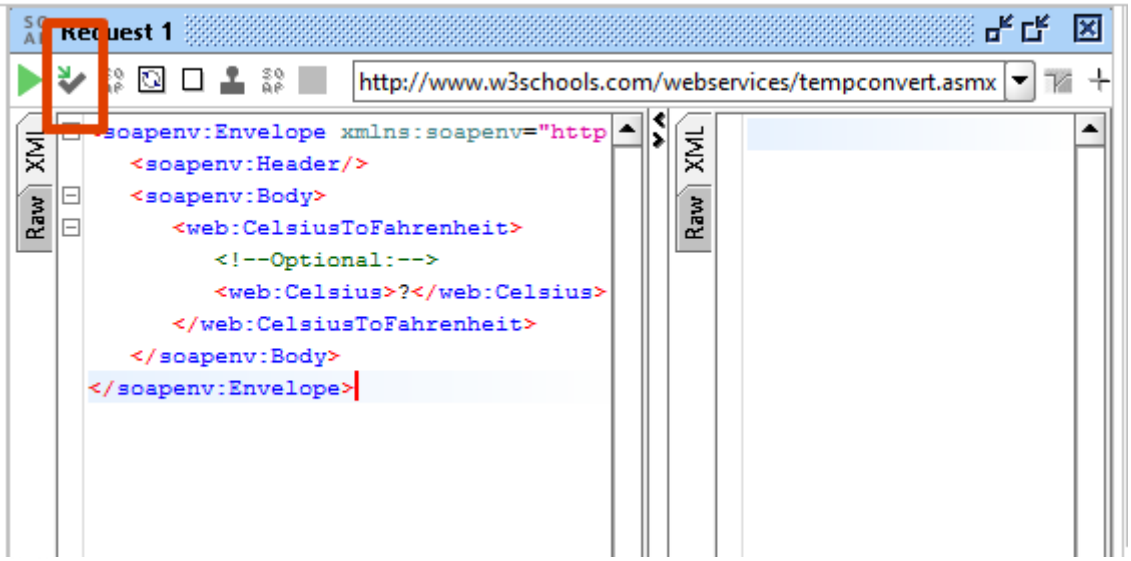
- WSDL
- EndPoint

En la pestaña **WS-I Compliance**, se puede validar si el descriptor WSDL cumple con el estándar **WS-I Basic Profile** de interoperabilidad del **Web Services Interoperability Organization (WS-I)**.

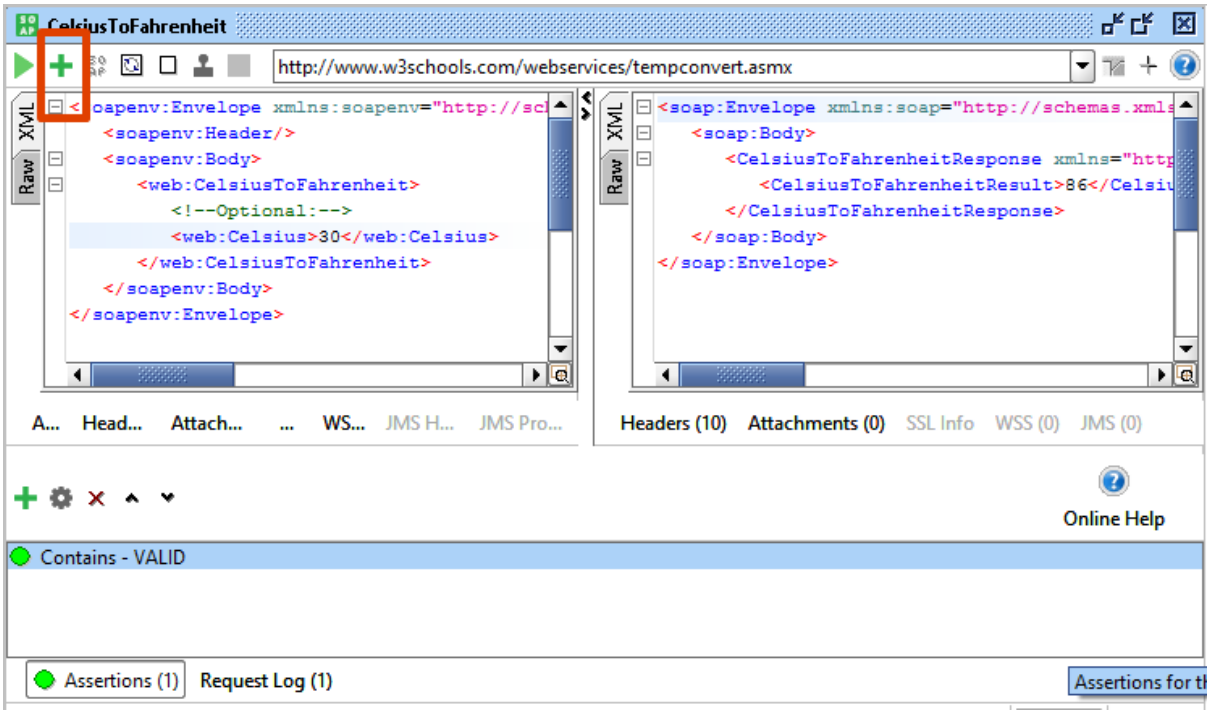
Pinchando doblemente en la Request, se abre una ventana que permite realizar la petición, solo hay que sustituir las ? por datos y pinchar sobre el triangulo verde de la esquina superior izquierda para ejecutarlo.



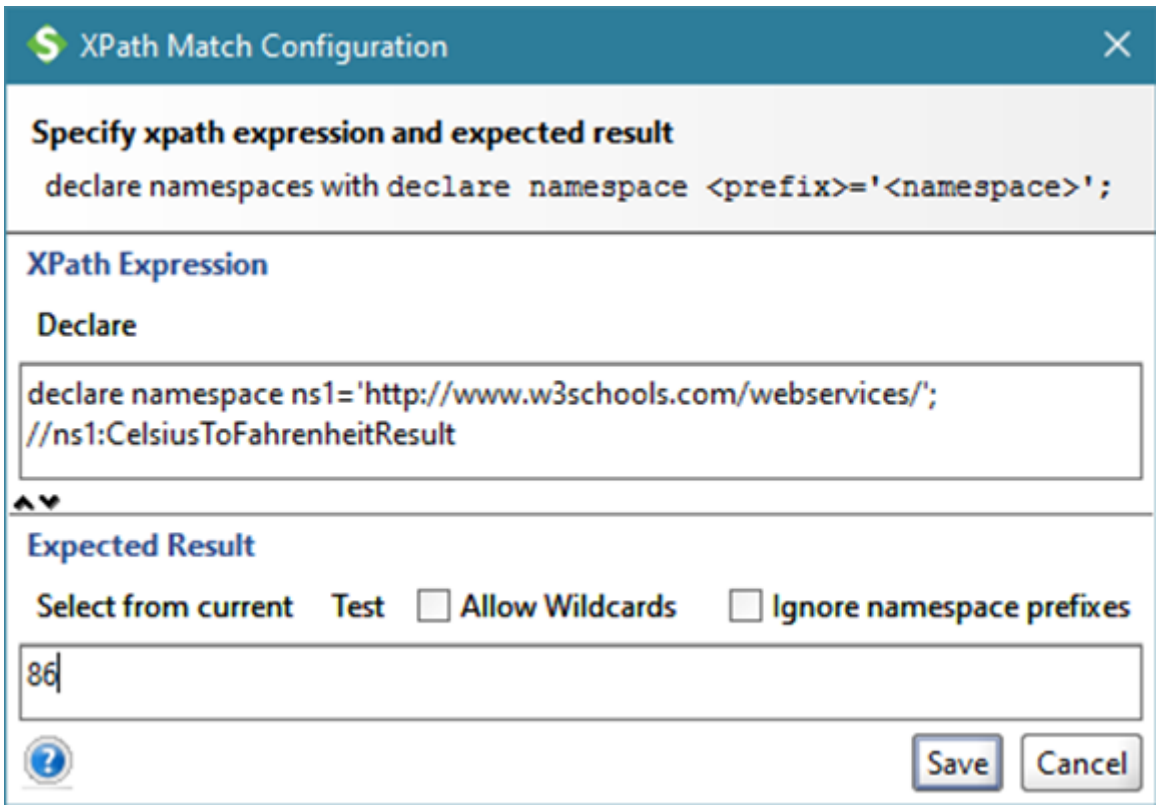
La ejecución de la petición, representa la ejecución de la logica aprobar, todavia no se han definido las validaciones (aserciones), para ello se dispone de un botón para crear un test de esa request



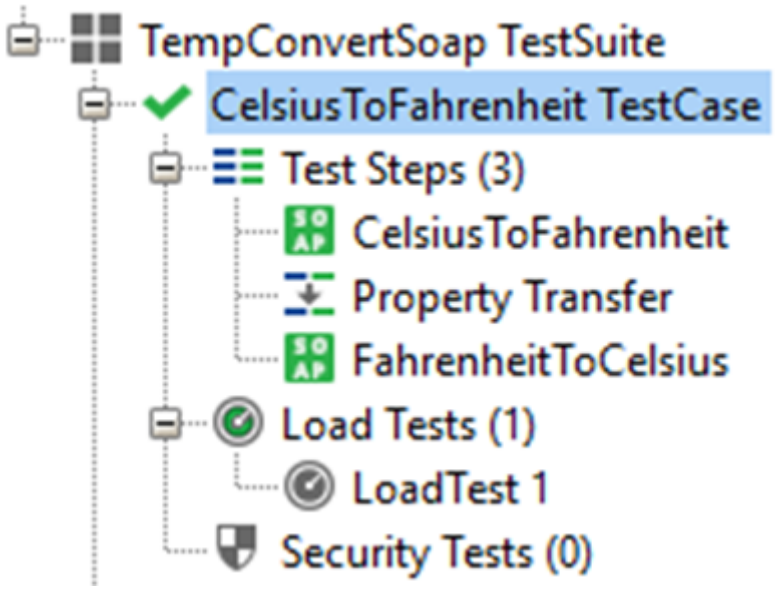
Abriendo el Test, se tiene la posibilidad de añadir una aserción.



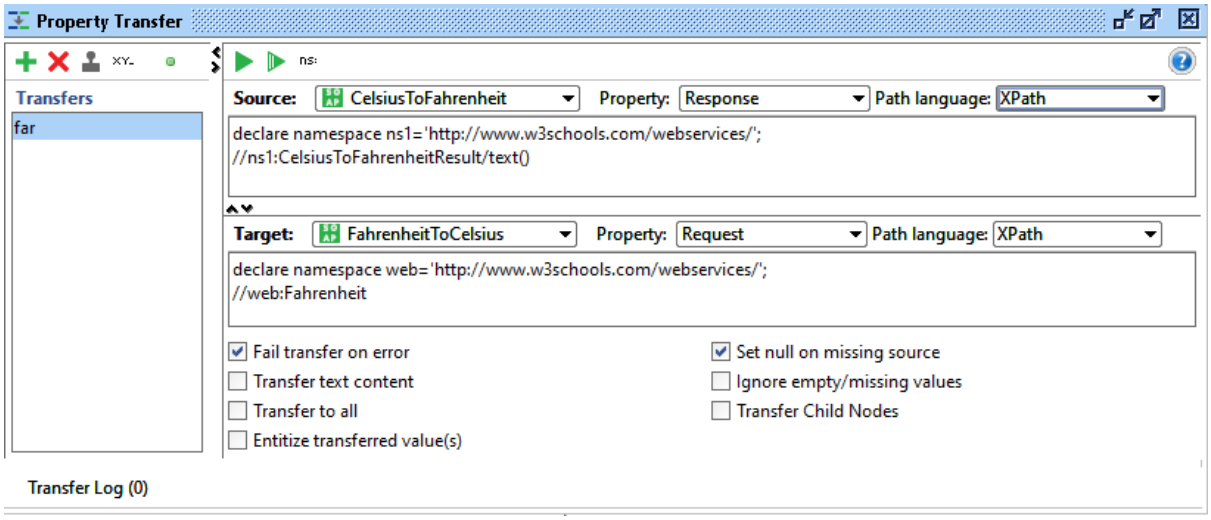
Una posible Aserción seria una expresión Xpath.



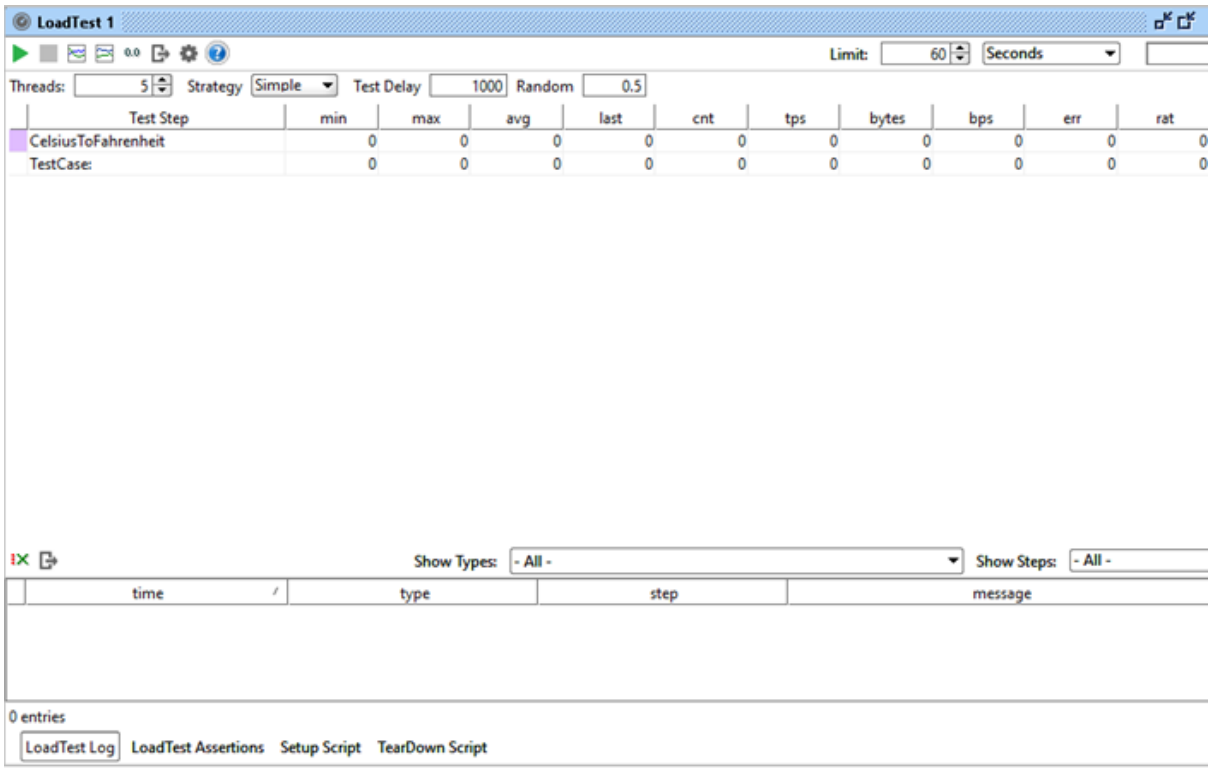
Se pueden pasar datos (Properties) entre distintos Step de un TestCase, para ello hay que crear un Step, del tipo Property Transfer.



En ella se ha de seleccionar, por ejemplo con Xpath el valor a trasladar de un Step al siguiente y el lugar dentro del siguiente Step para situar el valor.



También se pueden crear LoadTest (Test de carga)



5.4. SoapUI Maven Plugin

Plugin para poder lanzar los proyectos SoapUI de forma automatizada con Maven.

Se necesita añadir un repositorio para poder descargar el plugin, ya que no se encuentra en el repositorio central de Maven.

```
<pluginRepositories>
    <pluginRepository>
        <id>SmartBearPluginRepository</id>
        <url>http://www.soapui.org/repository/maven2/</url>
    </pluginRepository>
</pluginRepositories>
```

Y el Plugin

```
<plugin>
    <groupId>eviware</groupId>
    <artifactId>maven-soapui-plugin</artifactId>
    <version>4.0.1</version>
    <dependencies>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.6</version>
        </dependency>
        <dependency>
            <groupId>eviware</groupId>
            <artifactId>maven-soapui-plugin</artifactId>
            <version>4.0.1</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.2</version>
        </dependency>
        <dependency>
            <groupId>xerces</groupId>
            <artifactId>xercesImpl</artifactId>
            <version>2.8.1</version>
        </dependency>
        <dependency>
            <groupId>commons-collections</groupId>
            <artifactId>commons-collections</artifactId>
            <version>3.2.2</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <phase>test</phase>
            <goals>
                <goal>test</goal>
            </goals>
            <configuration>
                <projectFile>TestConverterW3C-soapui-
project.xml</projectFile>
            </configuration>
        </execution>
    </executions>
</plugin>
```

NOTE	Ojo con los ficheros XML que se generan al guardar el proyecto de SoapUI, que son los que se leen desde el plugin de Maven, ya que incluyen en los XML de los mensajes SOAP un “/r” que habrá que eliminar, ya que sino no funcionarán los test.

6. JMeter

Herramienta independiente para realizar pruebas funcionales y de stress sobre un servidor Web.

Se pagina oficial con la documentación esta aquí

6.1. Que puede hacer

- Cuellos de botella en el sistema.
- Fallos en aplicaciones.
- Peticiones que es capaz de absorber el servidor.
- Simulación de un uso cotidiano de una aplicación.
- Simulación de estrés de una aplicación.
- . . .

6.2. Que no puede hacer

- JMeter simplifica la generación de los planes de Test, pero no puede generarlos por si mismo.
- Se precisa de tiempo para generar planes de Test eficientes.

6.3. Estructura

Al acceder a JMeter se ve que la aplicación esta dividida en dos partes.

- **Plan de Pruebas**. Donde desarrollaremos nuestros planes de pruebas.
- **Banco de Trabajo**. Donde tendremos las herramientas y operaciones a utilizar en nuestro plan de pruebas.

Podremos mover contenidos del Banco de Trabajo a los Planes de Prueba para su uso.

6.4. Instalación

La descarga se realiza desde aquí

Necesaria JVM. Compatibilidad 1.5x en adelante.

Para asignar memoria a JMeter, emplearemos la variable de entorno JVM_ARGS.

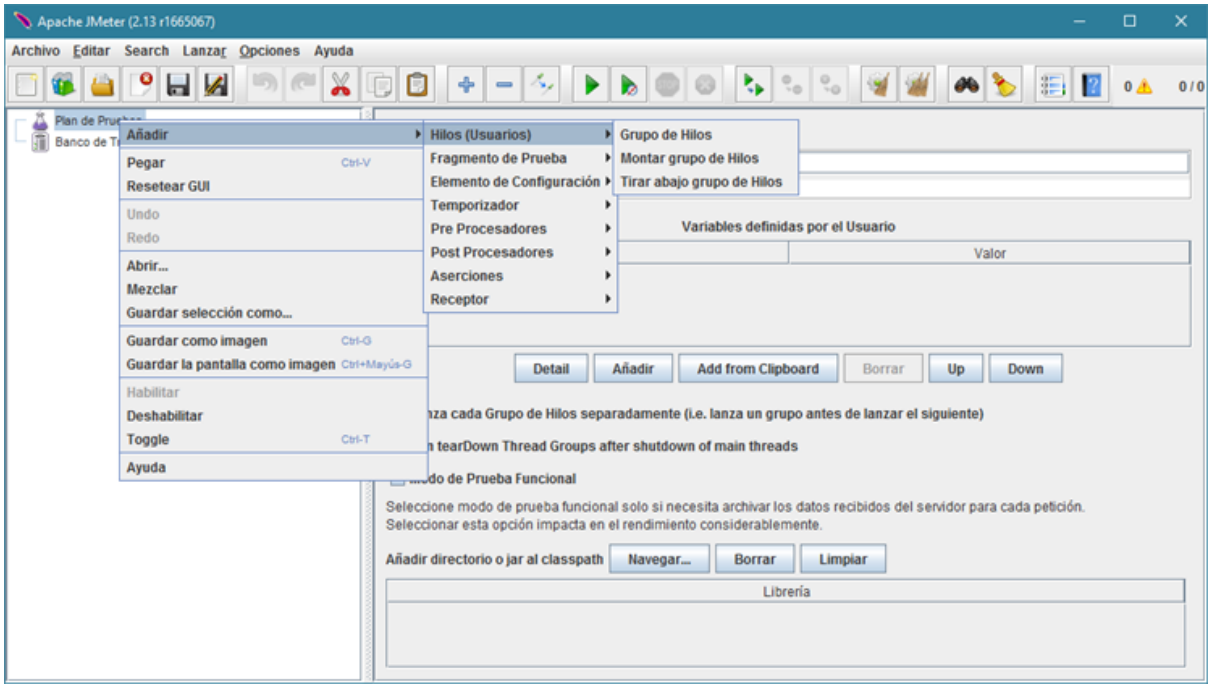
```
JVM_ARGS=-Xms256m -Xmx512m
```

Para incluir un jar en nuestros test, por ejemplo el driver de una base de datos, se incluirá en la carpeta lib.

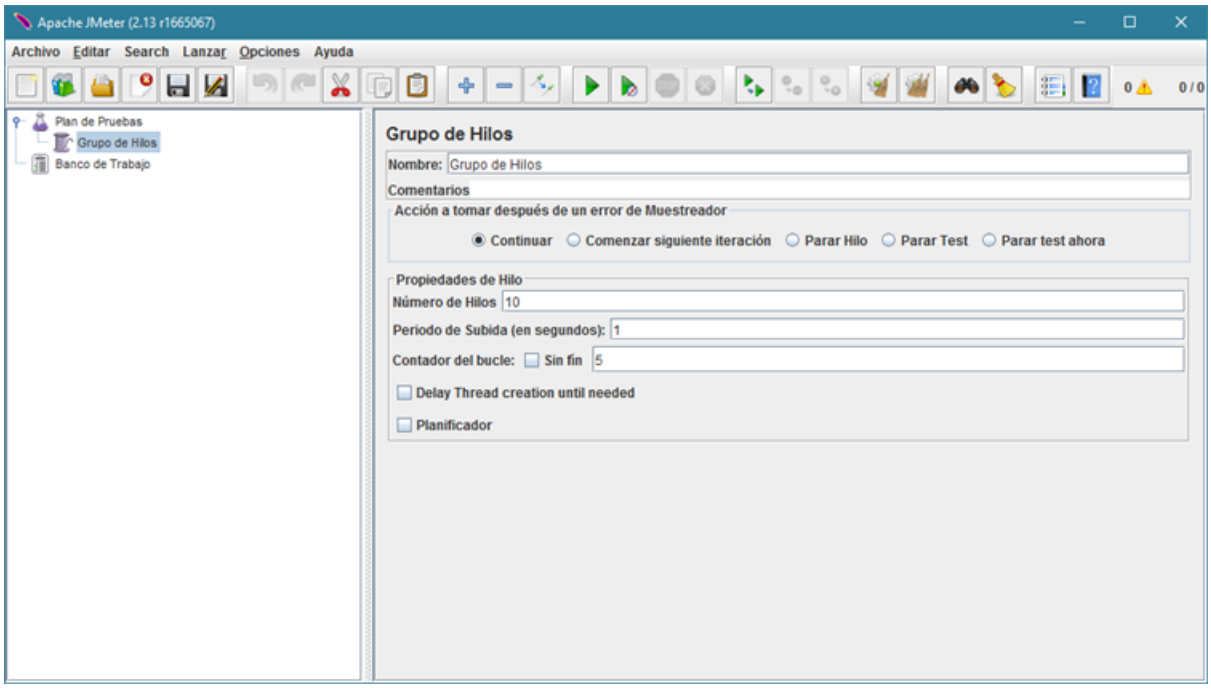
6.5. Plan de Pruebas

Los Planes de Pruebas estan compuestos por elementos de las siguientes tipologías.

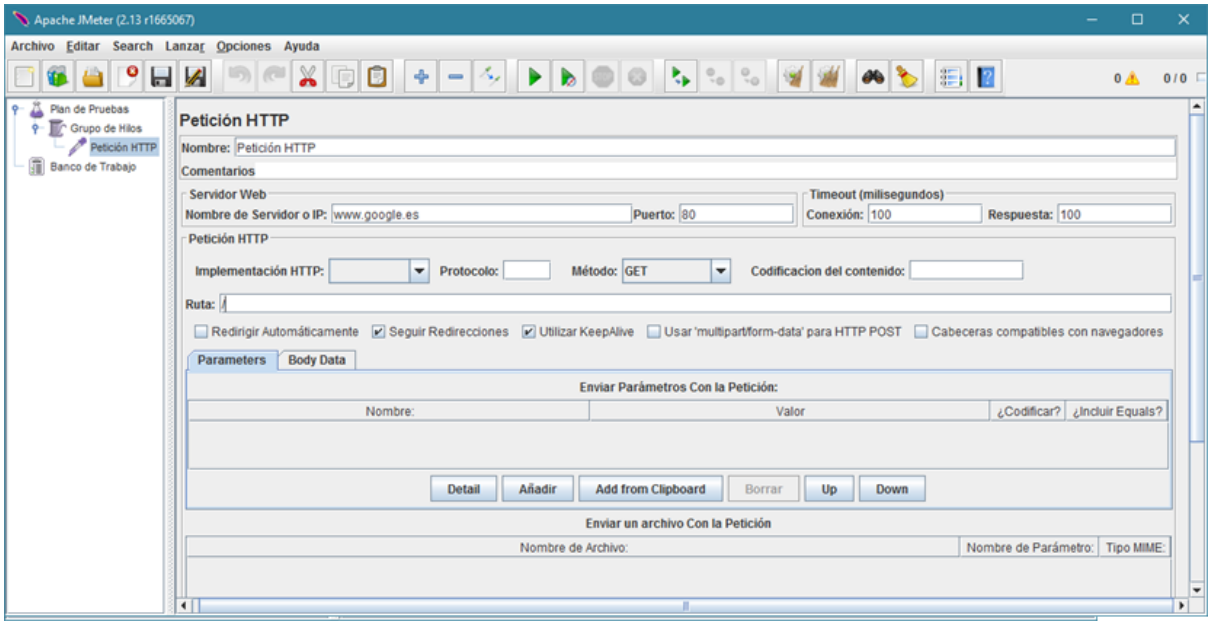
- Grupo de Hilos. Simulara al numero de usuarios.



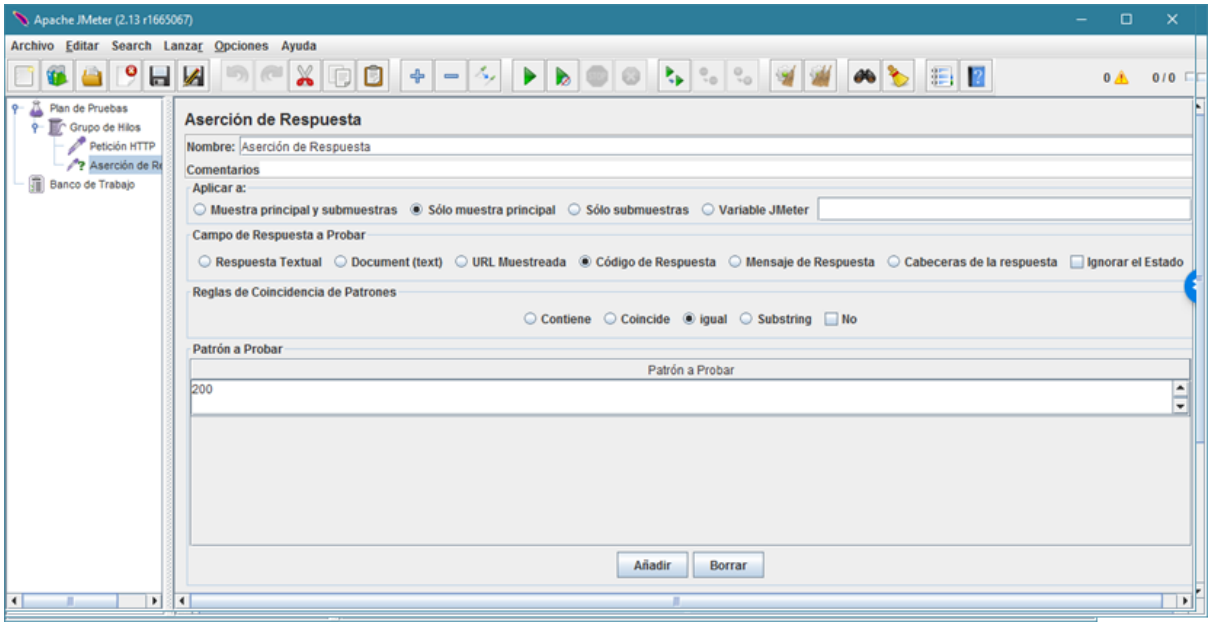
Permite definir usuarios simultáneos y peticiones de cada usuario.



- Procesadores. Permiten realizar modificaciones en la petición original. Se dividen en
  - ¬ Pre-Procesadores. Modifican la petición antes de ejecutarse.
  - ¬ Post-Procesadores. Modifican la petición después de ejecutarse.
- Controladores. Existen de diversos tipos
  - ¬ Muestreadores. Indican las acciones que JMeter puede hacer. Tenemos entre otros los siguientes tipos:
    - ¬ HTTP Request. Similar a HTTP Request Defaults en su definición, pero no representa un configuracion de una peticion HTTP, sino que realiza una. Ademas permite definir fichero a enviar.



- Petición JDBC. Nos permite ejecutar una consulta sobre la Base de Datos.
- Petición WebServices (SOAP). Permite realizar peticiones SOAP a un servicio web, empleando el WSDL.
    - ¬ Aserciones. Comprobación de resultados esperados. Algunas de las aserciones disponibles son
- Aserción de respuesta. Nos permite comprobar si alguno de los campos de la respuesta coincide con un determinado patrón, OJO!! es la respuesta no el HTML.



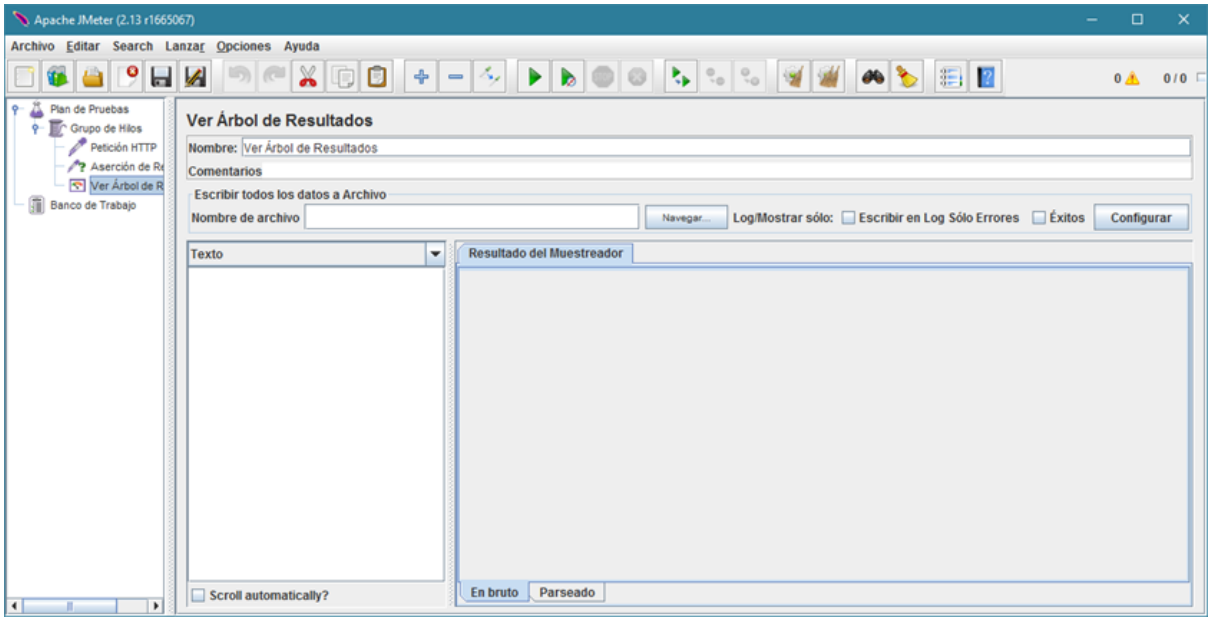
- Aserción de esquema XML. Valida que la respuesta cumple el esquema indicado mediante un fichero XSD.
- Aserción XML. Comprueba que el resultado es un XML bien construido.
- Aserción de Tamaño. Nos permite comprobar si alguno de los campos de la respuesta tiene un tamaño determinado.
    - ¬ Elementos de configuración. Trabaja en conjunto con los Muestreadores para modificarlos.
- HTTP Request Defaults. Permite definir la configuracion base para las peticiones HTTP que se realicen en el grupo de hilos. SE puede definir: Method HTTP, URL, PORT, PATH, ENCODING, URL PARAMETERS, BODY, TIMEOUT de conexion y respuesta, conexion a traves de proxy.
- HTTP Cookie Manager. Permite que las peticiones realizadas en un mismo hilo, compartan las Cookies.
    - ¬ Controladores lógicos. Modifican la lógica de lo que se debe hacer (Toma de

decisiones).

¬ Temporizadores. Incluye pausas en el test, para simular la realidad.

¬ Receptores o Listeners. Muestran los resultados de las peticiones en distintos formatos. Para mostrar los resultados se debe añadir un Listener al plan de pruebas. Tenemos entre otros los siguientes tipos:

¬ Árbol de resultados. Para cada petición muestra la respuesta HTTP, la petición y los datos HTML devueltos.



- Informe agregado. Muestra un resumen de los resultados
- Gráfico de resultados. Muestra un gráfico de rendimiento

6.6. Banco de trabajo

El Banco de Trabajo es el lugar donde tengamos nuestras herramientas que nos ayudaran a configurar nuestro Plan de Pruebas, una de las herramientas mas interesantes que tendremos será el **Servidor Proxy HTTP**, esta herramienta nos permitirá obtener los pasos seguidos en una navegación, es decir es capaz de traducirnos a acciones de JMeter, los pasos que hacemos en una navegación, para posteriormente poder grabarlos como macro de JMeter.

Los que se genera en el **banco de trabajo**, solo esta disponible mientra JMeter esta arrancado.

Para utilizar el **Servidor Proxy HTTP**, debemos seguir los siguientes pasos:

- Creamos en el Banco de Trabajo un elemento Servidor Proxy HTTP.

- Configuramos el controlador objetivo, es decir donde queremos que vaya poniendo los pasos que se vayan a seguir en la navegación.
- Establecemos los patrones a incluir o excluir en la navegación, es posible que no interese que se almacenen imágenes, javascript, …
- Arrancamos el Proxy.
- Configuramos el navegador a utilizar, para que pase a través de este proxy.
- Realizamos la navegación.
- Paramos el Proxy.

6.7. Jmeter Maven Plugin

Este plugin permite ejecutar los test generados con JMeter en una fase de Maven, la documentación se puede encontrar aquí

Se ha de añadir el plugin, seleccionando la fase en la que se quiere ejecutar, las habituales serán **integration-test** y **verify**.

```
<plugin>
    <groupId>com.lazerycode.jmeter</groupId>
    <artifactId>jmeter-maven-plugin</artifactId>
    <version>1.10.1</version>
    <executions>
        <execution>
            <id>jmeter-tests</id>
            <phase>verify</phase>
            <goals>
                <goal>jmeter</goal>
            </goals>
       </execution>
    </executions>
</plugin>
```

Además se han de incluir los ficheros **jmx** generados con JMeter en el directorio **<Project Dir>/src/test/jmeter**

Se pueden añadir plugins a la ejecución

```
<plugin>
    <groupId>com.lazerycode.jmeter</groupId>
    <artifactId>jmeter-maven-plugin</artifactId>
    <version>1.10.1</version>
    <executions>
        <execution>
            <id>jmeter-tests</id>
            <phase>verify</phase>
            <goals>
                <goal>jmeter</goal>
            </goals>
            <configuration>
                <jmeterPlugins>
                    <plugin>
                        <groupId>kg.apc</groupId>
                        <artifactId>jmeter-plugins</artifactId>
                    </plugin>
                </jmeterPlugins>
            </configuration>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>kg.apc</groupId>
            <artifactId>jmeter-plugins</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>
</plugin>
```

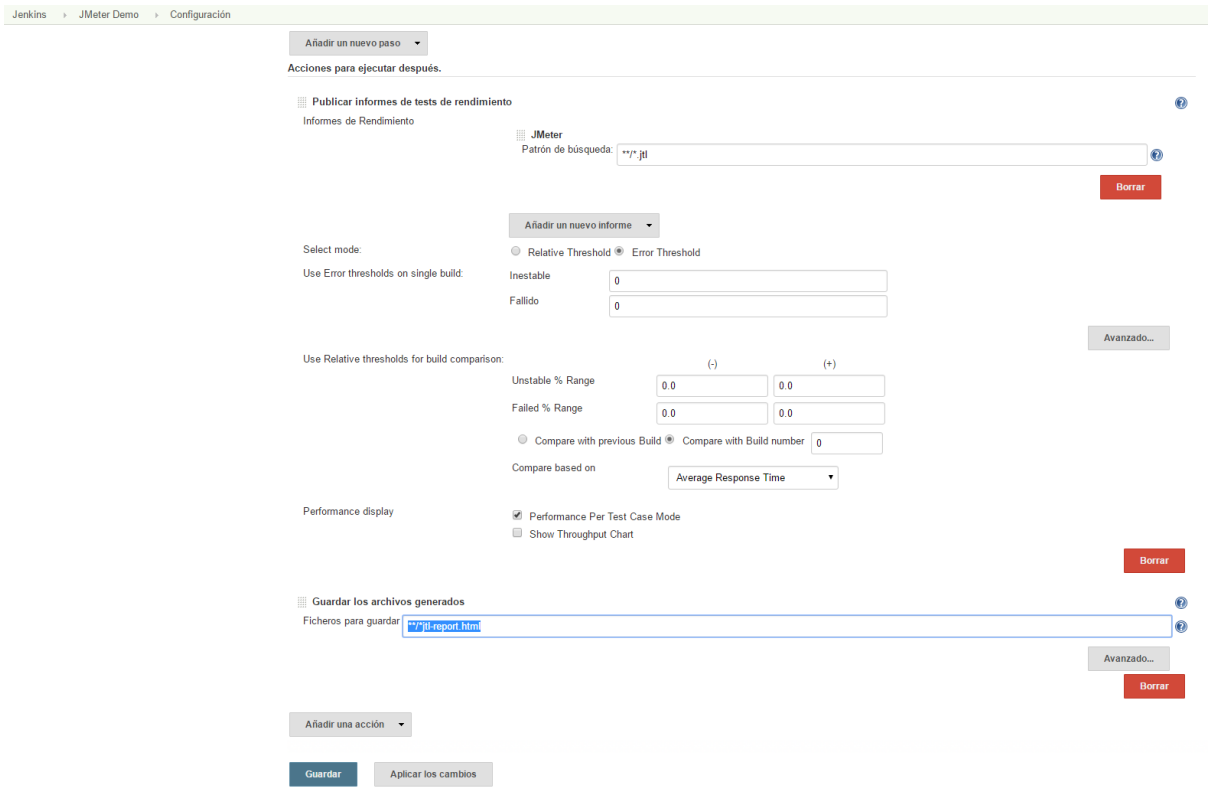
Se puede encontrar una aplicación de ejemplo aquí

6.8. Jenkins Performance Plugin

Es un plugin de Jenkins, que permite incorporar los resultados obtenidos con JMeter en la UI de Jenkins.

Una vez añadido, habra que definir un nuevo **Post Procesor** a la tarea de tipo **Publicar Informes de Test de rendimiento**, donde se han de establecer **patrón de búsquedas** a ***/**.jtl

Y posteriormente otro nuevo **Post Procesor** de tipo **Guardar los archivos generados**, donde se han de establecer **Ficheros para guardar** a **/*jtl-report.html



Una vez se lanza la tarea, se obtienen nuevos datos en la vista de la tarea como **Tendencia de rendimiento**