Escuela Tècnica Superior de Ingeniería Informàtica

Universidad Politècnica de València

TESTING EN SERVICIOS ONLINE Y LOCALES

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Zaquiel Rodríguez Arce

Tutor: Carlos Carrascosa Casamayor

2016-2017

1. Resumen

Este trabajo explica de forma completa cada una de las fases del proceso del *testing* donde se indican: los activos, las validaciones que se necesitan en el proceso y cómo se deben hacer. Además, se señalan los aspectos más importantes de cada uno de ellos y en qué ha de fijarse el tester cuando los esté realizando. Finalmente, este trabajo explica el objetivo y la importancia de cada una de las fases acabando con una comparación entre los servicios y una breve conclusión sobre el *testing*.

Palabras clave: activos, *testing*.

2. Abstract

This work explains each phases of the testing process in a full way. They indicate assets and validations which the process needs and how to do them. In addition, this work point to the most important aspects of each of them and where the tester must focus while he is doing them. Finally, it explains the target and valuable of each phase finishing with a contrast between services and a small conclusion about testing.

Palabras clave: assets, testing.

Agradecimientos

En primer lugar, me gustaría agradecer a mi tutor Carlos Carrascosa Casamayor por su dedicación, tiempo y sus consejos durante todo el proyecto.

Quisiera agradecer a la Universidad Politécnica de Valencia por aceptarme como alumno y haberme dado la valiosa oportunidad de realizar prácticas en Sopra Steria, una consultoría que, con el respaldo de la universidad, me enseñó y me introdujo en el mundo del testing.

Por último, pero no por ello menos importante, doy gracias a mis familiares y compañeros que me han apoyado en todo momento.

Gracias

Acrónimos

DTAN	Documento técnico de alto nivel
DTD	Documento técnico detallado
PLP	Plan de pruebas
ETF	Entrada funcional
DAT	Datos técnicos
SCF	Script funcional
DAR	Documento de alto rendimiento
ETR	Entrada de rendimiento
SCR	Script de rendimiento
SCS	Script de seguridad
WSDL	Web services description language
XML	Extensible markup language
CSV	Comma-separated values
JDBC	Java Database Connectivity
FW	Framework
STD	Standalone
SDB	Saving database
SSH	Secure shell

Tabla de contenidos

1. **Motivación y objetivos** 6
2. **Introducción** 6
3. **Tecnologías y herramientas** 7
4. **Proceso del testing** 14
   4.1. Preproceso del testing 16
   4.2. Documento técnico de alto nivel (DTAN) 17
      4.2.1. *Fase de validación* 18
   4.3. Documento técnico detallado (DTD) 22
      4.3.1. *Validación del DTD* 22
   4.4. Fase de pruebas funcionales 22
      4.4.1. Plan de pruebas (PLP) 23
         4.4.1.1. *Ejemplo explicativo* 24
         4.4.1.2. *Fase de validación* 24
      4.4.2. Entrada funcional (ETF) [FW2, FW3] 27
         4.4.2.1. *Ejemplo explicativo* 28
      4.4.3. Entrada funcional (ETF) [Standalone] 29
         4.4.3.1. *Ejemplo explicativo* 29
      4.4.4. Datos técnicos (DAT) [FW online, STD público] 30
         4.4.4.1. *Ejemplo explicativo* 31
      4.4.5. Datos técnicos (DAT) [Batch, Standalone] 32
         4.4.5.1. *Ejemplo explicativo (caso real)* 33
      4.4.6. Activo WSDL [FW2, FW3, Standalone] 33
         4.4.6.1. *Ejemplo explicativo* 34
      4.4.7. Script funcional (SCF) [FW2, FW3, STD público] 35
      4.4.8. Script funcional (SCF) [STD público] 39
      4.4.9. Script funcional (SCF) [STD privado] 41
      4.4.10. Ejecución funcional (BATCH) 44
   4.5. Fase de pruebas de seguridad 46
      4.5.1. Script de seguridad (SCS) [FW online, STD público] 46
   4.6. Fase de rendimiento 48
      4.6.1. Documento de alto rendimiento (DAR) [FW2, FW3, STD público] 51
         4.6.1.1. *Ejemplo explicativo* 52
      4.6.2. Entrada de rendimiento (ETR) [FW2, FW3, STD público] 53
         4.6.2.1. *Ejemplo explicativo* 53
      4.6.3. Script de rendimiento (SCR) [FW2, FW3, STD público] 54
         4.6.3.1. *Ejemplo explicativo* 56
      4.6.4. Script de rendimiento (SCR) [STD privado] 57
   4.7. Entrega y reflexiones del testing 58
5. **Caso práctico real** 59
6. **Optimización del proceso del testing** 81
7. **Conclusiones** 83
8. **Bibliografía** 85

1. Motivación y objetivos

El objetivo de este trabajo es informar y enseñar al lector el proceso que siguen las empresas y el estándar de factoría que se ha de seguir para asegurar un servicio de calidad. Además, durante el trabajo se hace hincapié en los datos más sensibles, así como en los aspectos más importantes del servicio debido a que un mal funcionamiento en alguno de estos apartados llevaría el servicio al fin de su vida útil.

Mi entrada al mundo laboral se produjo gracias a las prácticas de la universidad en una empresa internacional centrada en el mundo del *testing* (proceso de pruebas para testear una aplicación y comprobar si el servicio a testear es de calidad o no). A pesar de usar todo tipo de servicios del mundo de la informática a diario, hasta que no entré en la empresa no comprendía, ni me imaginaba, todo el esfuerzo y el trabajo que se encontraba detrás de cada uno de los servicios para que funcionaran correctamente y tuvieran éxito a nivel comercial.

Este trabajo también pretende difundir la importancia del *testing* en la actualidad, así como reconocer y valorar este proceso tan esencial e imprescindible.

Por último, se concluirá con una reflexión sobre el *testing* y una breve comparación entre el proceso que se sigue con los diversos frameworks que aparecen y se explican durante el trabajo ([FW2, FW3] en servicios online y [Standalone, BATCH] en servicios locales).

2. Introducción

Hoy en día, gran parte de nuestro mundo está relacionado con la informática y ésta con los servicios. Desde una página web de una empresa hasta el peso en un supermercado, usan servicios e, independientemente de si son servicios online (FW2 y FW3) o locales (Standalone y BATCH), deben de funcionar de forma correcta si quieren sobrevivir en un mercado y un mundo cada vez más competitivo. El objetivo de este proyecto es informar de las herramientas, documentos, pruebas y los pasos que se han de seguir para testear un servicio (proceso inalterable que cumple con el estándar de factoría).

El proceso del *testing* es el método que realiza un tester para asegurarse de que el software que se está testeando es de calidad. Este proceso sigue unos pasos y técnicas muy concretas (acordes con los pasos que realizan las consultorías en la actualidad para testear un servicio) y esa calidad depende de su funcionalidad, rendimiento y de su nivel de seguridad.

El *testing* cada vez cobra una mayor importancia debido a un mundo cada vez más globalizado. En la actualidad, hay muchos softwares similares y su calidad puede marcar la diferencia entre el éxito o el fracaso. Esa calidad que se exige durante el *testing* obliga al software a que:

- Todas las funciones respondan correctamente y no salte ningún error que no haya provocado el usuario (el usuario puede escribir un nombre incorrecto al registrase o tratar de pedir un producto sin existencias).
- No se produzca ningún error de seguridad como solicitudes del usuario incorrectas o errores con relación a datos personales.
- Rinda adecuadamente ante picos de actividad y tenga tiempos de respuesta cortos (este aspecto es uno de los más importantes ya que los usuarios están acostumbrados en que el software responda en menos de un segundo).

Durante el proyecto se explica cada activo que va a ser necesario para el proceso del *testing* y finaliza con una breve comparación entre los servicios y una conclusión del trabajo.

Con este proyecto se espera mostrar al usuario el trabajo y el esfuerzo que se esconde detrás de un servicio de calidad. Además, este trabajo busca enseñar los pasos que se han de seguir dentro del mundo del *testing* (cumpliendo con el estándar) si se desea testear un software propio o de alguna empresa. Para esto, el proyecto hace uso tanto de casos creados especialmente para este proyecto como de casos reales (estos casos provienen de una empresa de consultoría, por lo tanto, cada uno de los casos reales mostrados han sido implantados en el mundo real y se ha confirmado su calidad en todos los aspectos y ámbitos del *testing*).

3. Tecnologías y herramientas

El mundo de la informática se está actualizando constantemente y cada año surgen programas que sustituyen a otros o los vuelven obsoletos. En el *testing* esto también sucede y en gran medida es porque siempre se busca el *testing* perfecto y de la forma más automatizada posible.

El proceso del *testing* requiere de diversas herramientas que pueden variar según el framework y que el tester las usará de una determinada forma según en la fase del *testing* en la que se encuentre.

Durante este proyecto, el *testing* hace uso principalmente de la herramienta *SoapUI* [2]. Esta herramienta es una de las más comunes actualmente y a cada año que pasa adquiere más adeptos, esto se debe: a su amplio abanico de posibilidades, su nivel de personalización y su compatibilidad con otros programas como el *Hermes* [11] (se tratará esta herramienta durante la fase funcional del *testing*).

Otra de las herramientas que tiene un gran peso en el sector del *testing* es el *Jmeter* [14] y, al igual que el *SoapUI*, admite extensiones y personalizaciones por parte del tester para mejorar el proceso.

Durante el proyecto se trata las interacciones que puede tener el servicio con la base de datos. Estas interacciones se realizan con el *SQLDeveloper* [9], una de las principales herramientas que existen en el mercado y en el mundo del *testing* con relación a las bases de datos (en este apartado se ha realizado una breve comparación con otra herramienta importante en éste ámbito, la herramienta *Toad* [17]).

Por último, debido a los servicios que hacen uso de uno de los framework locales, se explica y visualiza la herramienta Putty y su implicación en el proceso del *testing*.

3.1. SoapUI vs Jmeter

Ambas herramientas son capaces de ejecutar los tres tipos de pruebas que se requieren para el *testing* (funcional, rendimiento y seguridad) pero *Jmeter* tiene dos grandes problemas que hace que pierda terreno frente al *SoapUI*. Estos problemas se pueden clasificar desde dos puntos de vista: el punto de vista de la empresa y el del cliente.

- ✓ Desde el punto de vista de una consultoría, el Jmeter (véase la *figura 2*) tiene una mayor complejidad y una interfaz poco amigable (hay que crear un lanzador si no se desea ejecutar mediante el terminal). Esto supone que las empresas se decanten por el *SoapUI* (véase la *figura 1*) y así reducir el tiempo de aprendizaje de sus empleados y empezar más rápido con la personalización de la herramienta mediante scripts y activos propios de la empresa o del tester.

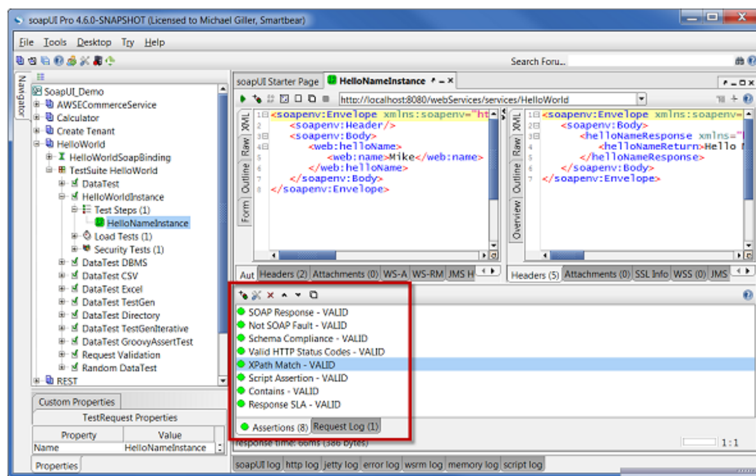


***Figura 1**. Interfaz de la herramienta SoapUI*

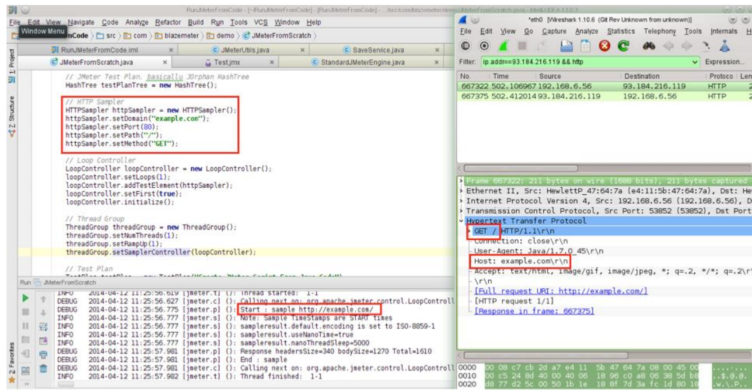
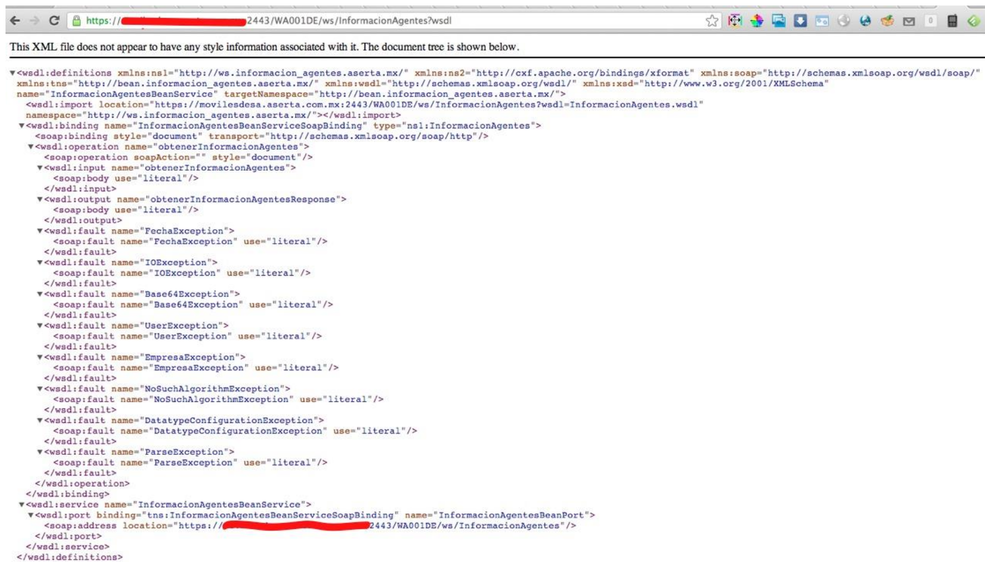


Figura 2. *Interfaz de la herramienta Jmeter*

✓ Desde el punto de vista del cliente, el *Jmeter* (véase la *figura 3*) no aporta ninguna facilidad, ni al cliente (para realizar algunos activos necesarios en el *testing*, es común que el tester pida ayuda al cliente debido a los profundos conocimientos que debe tener el tester para realizarlo, esto se debe a que el *Jmeter* a diferencia del *SoapUI*, no genera una plantilla) ni al tester, ya que al no ofrecer ningún tipo de plantilla, el tester ha de seguir unos pasos considerablemente elaborados con gran cantidad de datos, datos que el *SoapUI* (véase la *figura 4*) automáticamente recopila y genera.



https://2443/WA001DE/ws/InformacionAgentes?wsdl

This XML file does not appear to have any style information associated with it. The document tree is shown below.


```
<wsdl:definitions xmlns:ns1="http://ws.informacion_agentes.aserta.mx/" xmlns:ns2="http://cxf.apache.org/bindings/xformat" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://bean.informacion_agentes.aserta.mx/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="InformacionAgentesBeanService" targetNamespace="http://bean.informacion_agentes.aserta.mx/">
  <wsdl:import location="https://movilesdesa.aserta.com.mx:2443/WA001DE/ws/InformacionAgentes?wsdl=InformacionAgentes.wsdl" namespace="http://ws.informacion_agentes.aserta.mx/"></wsdl:import>
  <wsdl:binding name="InformacionAgentesBeanServiceSoapBinding" type="ns1:InformacionAgentes">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="obtenerInformacionAgentes">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="obtenerInformacionAgentes">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="obtenerInformacionAgentesResponse">
        <soap:body use="literal"/>
      </wsdl:output>
      <wsdl:fault name="FechaException">
        <soap:fault name="FechaException" use="literal"/>
      </wsdl:fault>
      <wsdl:fault name="IOException">
        <soap:fault name="IOException" use="literal"/>
      </wsdl:fault>
      <wsdl:fault name="Base64Exception">
        <soap:fault name="Base64Exception" use="literal"/>
      </wsdl:fault>
      <wsdl:fault name="UserException">
        <soap:fault name="UserException" use="literal"/>
      </wsdl:fault>
      <wsdl:fault name="EmpresaException">
        <soap:fault name="EmpresaException" use="literal"/>
      </wsdl:fault>
      <wsdl:fault name="NoSuchAlgorithmException">
        <soap:fault name="NoSuchAlgorithmException" use="literal"/>
      </wsdl:fault>
      <wsdl:fault name="DatatypeConfigurationException">
        <soap:fault name="DatatypeConfigurationException" use="literal"/>
      </wsdl:fault>
      <wsdl:fault name="ParseException">
        <soap:fault name="ParseException" use="literal"/>
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="InformacionAgentesBeanService">
    <wsdl:port binding="tns:InformacionAgentesBeanServiceSoapBinding" name="InformacionAgentesBeanPort">
      <soap:address location="https://2443/WA001DE/ws/InformacionAgentes"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```


Figura 3. *Documento XML que hay que realizar para un proyecto de Jmeter*

```xml
<?xml version='1.0' encoding='ISO-8859-15'?><wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://
schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:ns1="http://schemas.xmlsoap.org/soap/http" name="
ED_ServicioCompleto">
    <wsdl:types>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://cxf.apache.org/arrays" attributeFormDefault="qualified"
elementFormDefault="qualified" targetNamespace="http://cxf.apache.org/arrays">
  <xsd:complexType name="Input_OperacionConsulta">
    <xsd:complexContent>
        <xsd:sequence>
          <xsd:element name="localeLanguage" type="xsd:string"/>
          <xsd:element name="localeCountry" type="xsd:string"/>
          <xsd:element name="localeVariant" type="xsd:string"/>
          <xsd:element name="codProveedor" nillable="true" type="xsd:string"/>
          <xsd:element name="codProducto" nillable="true" type="xsd:string"/>
          <xsd:element name="fechaEntrega" nillable="true" type="xsd:dateTime"/>
          <xsd:element name="fechaCaducidad" nillable="true" type="xsd:dateTime"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="Output_OperacionConsulta">
    <xsd:complexContent>
        <xsd:sequence>
          <xsd:element minOccurs="0" name="codError" nillable="true" type="xsd:string"/>
          <xsd:element minOccurs="0" name="desError" nillable="true" type="xsd:string"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Figura 4. *Documento XML que requiere el proyecto SoapUI.*

3.2. SQLDeveloper vs Toad

Como se ha mencionado anteriormente, otra de las herramientas que trata el proyecto es *SQLDeveloper*. Dicha herramienta, al igual que *Toad, ofrece un gran abanico de posibilidades para:*

- Crear archivos sql y mysql
- Establecer diversas conexiones de base de datos al mismo tiempo
- Múltiples pestañas y configuraciones que facilitan el manejo de la herramienta.

Las principales diferencias entre estas herramientas están relacionadas con el precio y algunas características propias.

- Mientras que *Toad* es una herramienta con un coste superior a los 1000 euros, *SQLDeveloper* es una herramienta gratuita. Este hecho sumado a que *SQLDeveloper* (véase en la *figura 5)* ofrece todo lo que un tester necesita para testear una aplicación hace que sea la primera opción para las consultorías (empresas dedicadas al *testing*).

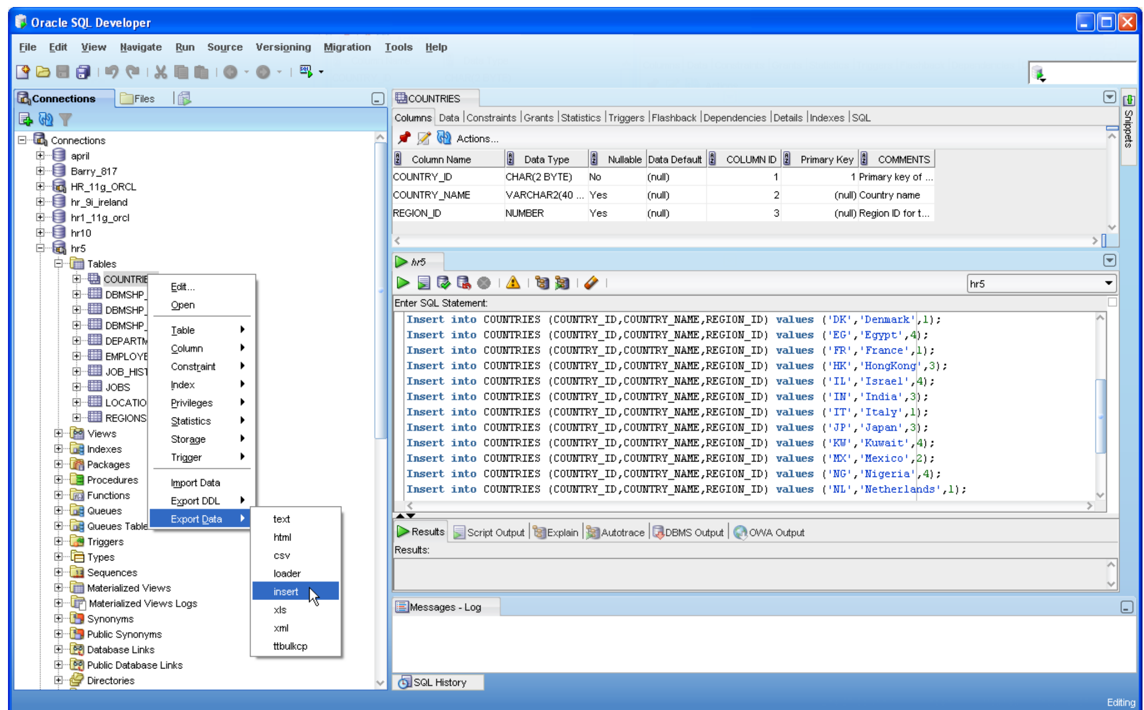


Figura 5. *Interfaz de la herramienta SQLDeveloper*

- *Toad* (véase en la *figura 6)* tiene características muy interesantes como diversas extensiones que agilizan el *testing* (compatible con muchas herramientas de testeo) y el uso de diagramas para visualizar las conexiones e interacciones entre las tablas de base de datos.
  La característica más relevante de *SQLDeveloper* es su mantenimiento, esto se debe a que Oracle lanza de forma continuada y frecuente actualizaciones de una gran calidad.

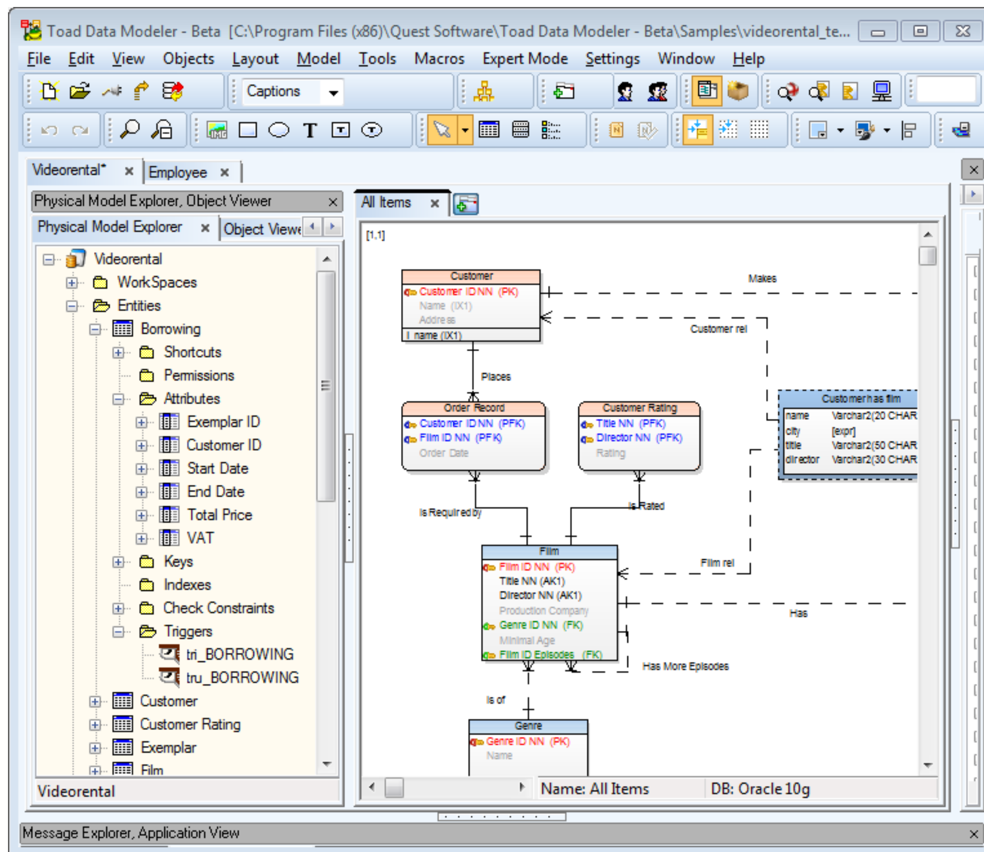


Figura 6. *Interfaz de la herramienta Toad*

3.3. Putty vs mobaXterm vs SmarTTY

Finalmente, este proyecto también tratará la herramienta *Putty* [12] para testear un servicio local. Esta herramienta suele ser corriente entre las consultorías, más en este apartado hay mucha diversidad de herramientas como *mobaXterm* [15] o *SmarTTY* [16] que también tienen un gran impacto en el mercado.

mobaXterm (véase en la *figura 7)* es una aplicación con opciones de pago y, aunque es mucho más completa que *Putty* y más flexible (puede conectarse de forma automática a casi cualquier cosa), tiene algunos problemas de rendimiento (algo extraño al ser de pago y un claro ejemplo de la importancia del *testing*, es decir, si se hubiera realizado un buen *testing* en la fase de rendimiento, no tendría estos problemas y quizás tuviera más éxito y fuera más popular).

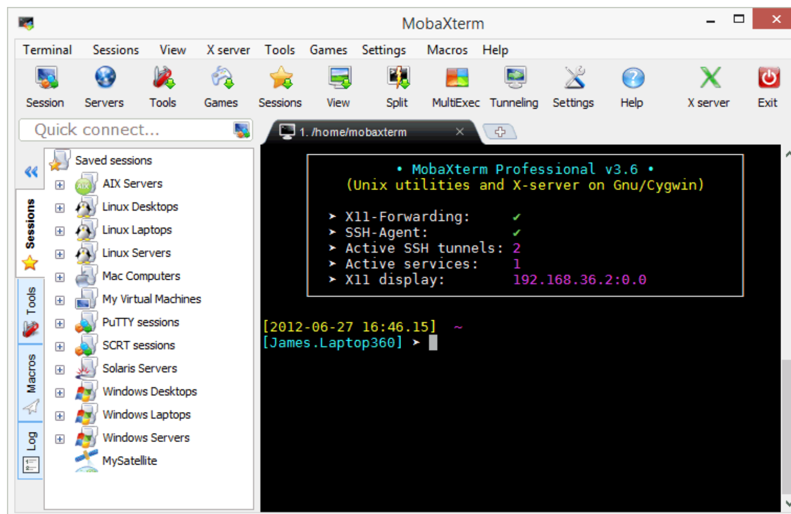


***Figura** 7. Interfaz de la herramienta mobaXterm*

SmarTTY (véase en la *figura 8)* es una aplicación en muchos aspectos superior a *Putty* ya que, siendo gratuito (como *Putty*), la aplicación acepta múltiples pestañas y es capaz de ejecutar aplicaciones gráficas (esto, a pesar de ser bueno, le agrega complejidad a la herramienta).

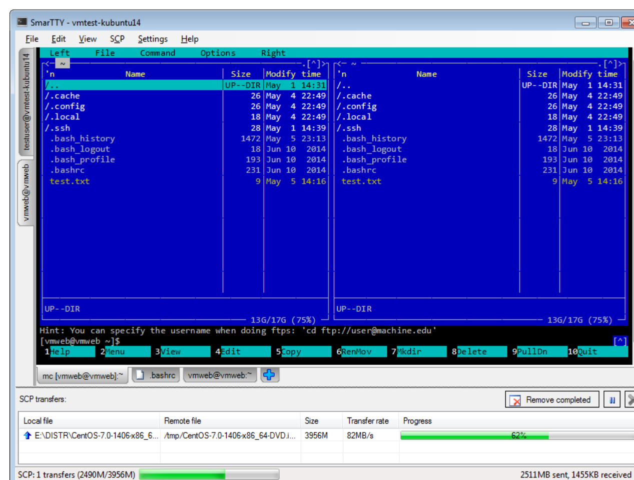


***Figura 8**. Interfaz de la herramienta SmarTTY*

La diferencia principal entre *Putty* (véase en la *figura 9)* y el resto de herramientas es su simplicidad. Su falta de opciones sumado con su personalización lo hace más atractivo para las empresas que desean personalizarlo con sus propios activos que facilitarán la detección de errores durante su ejecución (los activos lanzarán mensajes por la terminal durante su ejecución para que el tester sepa en todo momento que sección se está ejecutando).

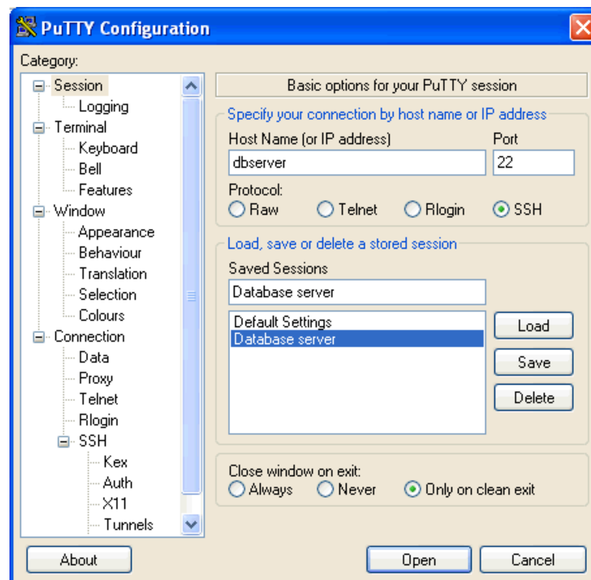


Figura 9. *Interfaz de la herramienta Putty*

4. Proceso del testing

El proceso del *testing* requiere, además de un proceso muy concreto (como se ha mencionado antes), de una serie de documentos muy concretos con sus correspondientes validaciones que ha de realizarse con mucho detalle y cuidado para que el proceso sea válido y correcto. Esto significa que el tester ha de tener experiencia y conocimientos profundos del proceso si desea realizar un *testing* de calidad.

El *testing* ha de comenzar con una previsión de cómo va a ser el proceso del *testing* y con una primera toma de contacto entre el tester y el servicio a testear (donde el cliente deberá de resolver todas las dudas iniciales del tester). Además de un acuerdo entre el cliente y el tester sobre el tiempo, la duración y lo informado que desee estar el cliente a lo largo del proceso.

Una vez realizado este preproceso del *testing*, el tester habrá de empezar revisando y validando los documentos base del servicio que le ha debido proporcionar el cliente al tester.

- ✓ **Documento técnico de alto nivel (DTAN):** Documento donde se definen todas las características y funciones que tiene el servicio.
- ✓ **Documento técnico detallado (DTD):** Documento que detalla el código que posee el servicio agrupado en clases y métodos.

Una vez establecido los documentos base, el proceso del *testing* se divide en tres fases:

1. **Testing desde el punto de vista funcional**: En este apartado se explica el proceso que ha de hacer el tester para verificar que el servicio funciona como se espera. Durante el proceso se explicarán los siguientes documentos/activos:

- ✓ **Plan de pruebas (PLP):** Activo con la lista de pruebas funcionales que ha de ejecutar el tester.
- ✓ **Entrada funcional (ETF):** Activo con una lista de entradas para el servicio (tantas entradas como casos funcionales que tiene el PLP).
- ✓ **Datos tecnicos (DAT):** Activo con las sentencias sql necesarias para llenar las tablas que requiere y usa el servicio.
- ✓ **WSDL:** Activo con los parámetros del servicio que permite al tester crear un proyecto SoapUI para testear el servicio.
- ✓ **Script funcional (SCF):** Activo que permite al tester ejecutar el servicio e, introduciendo una entrada, el tester será capaz de visualizar la salida que éste le devolvería.

2. **Testing desde la perspectiva de la seguridad**: En esta sección el tester ha de verificar si el servicio tiene implantado un código que proporcione un mínimo de nivel de seguridad, este nivel se establece en los estándares que ha de proporcionarte el cliente o estar indicado en el DTAN. En esta fase se explicará el SCS:

    - ✓ **Script de seguridad (SCS):** Proyecto donde el tester comprueba el nivel de seguridad de todos los parámetros de entrada del servicio.

3. **Fase de rendimiento**: En este caso se explican los documentos y el proceso a realizar por parte del tester para asegurarse de que el servicio responde correctamente tanto en situaciones estándar como en casos muy determinados (como en picos de actividad o estresando al servicio enviándole muchas peticiones de forma continuada). En la fase se encuentran unos pasos determinados en los que el tester debe realizar los siguientes documentos:

    - ✓ **Documento de alto rendimiento (DAR):** Activo que permite al tester llenar las tablas que el servicio necesite con cientos o miles de datos.
    - ✓ **Entrada de rendimiento (ETR):** Activo con cientos o miles de valores para cada uno de los parámetros del servicio.
    - ✓ **Script de rendimiento (SCR):** Proyecto que permite al tester comprobar el rendimiento del servicio ante diversos tráficos de usuarios.

Durante el trabajo se tratan las diversas variaciones que pueden sufrir los activos dependiendo del servicio que empleen (FW2, FW3, Standalone, Batch).

Finalmente, antes de la conclusión, se muestra el testing de un caso real que ha sido implantado con éxito en el mercado y cómo mejorar y optimizar el proceso. En este último punto hay que mencionar que el uso de nuevas tecnologías puede ayudar en gran medida, no sólo a acortar el tiempo del proceso sino a mejorarlo.

A continuación, se detallan los documentos anteriormente mencionados estructurados en sus respectivas fases y con sus correspondientes ejemplos.

4.1 Preproceso del testing

Antes de empezar con el *testing*, el tester tendrá que reunirse con el cliente y tratar diversos temas importantes que afectan directamente con el proceso del *testing*.

Cuando un cliente le proporciona un servicio al tester, quiere saber tres cosas:

- **Garantías**: el cliente quiere que el tester le garantice que cuando haya acabado, el servicio esté perfectamente listo para ser lanzado al mercado. Esto depende completamente del Tester y de que el *testing* que realice sea perfecto.

- **Precio**: si el tester es capaz de dar garantías al cliente, se tendrá que acordar un precio. El precio variará según: el tipo de servicio (un servicio online requiere de más activos y tiempo que un servicio local), el número de operaciones y su funcionalidad (al igual que no es lo mismo tener una operación que diez, también habrá que tener en cuenta otros aspectos: como el número de validaciones y caminos que tiene el servicio, así como si estas operaciones interactúan con la base de datos o no).

- **Información**: El cliente querrá saber en todo momento cómo va el proceso, sabiendo en cada fase por lo que está pagando y cómo responde su servicio a las pruebas y validaciones del *testing*.

Hitos Servicio	S1							S2						
	L	M	X	J	V	S	D	L	M	X	J	V	S	D
Gestión de Dudas y Validación de Petición														
Estimacion del tiempo de preparacion														
Estimación del tiempo de pruebas														
Negociacion del cliente con la estimacion del tiempo														
el Tester recibe el DTD y el WSDL														
Preparacion Plan de Pruebas y Datos del Sanity														
Estimación Definitiva del Tiempo														
Se informa al cliente del tiempo y se le proporciona el sanity														
Preparacion plan de Pruebas Completo														
Preparacion pruebas funcionales														
Ejecucion pruebas funcionales														
Preparacion pruebas Rendimiento														
Ejecucion pruebas Rendimiento														
Preparacion y ejecucion de pruebas de seguridad														
Entrega y cierre de la Petición														

Figura 10. *Todos los pasos que ha de realizar el tester para testear un servicio*

En la *figura 10* se observa todos los pasos que ha de realizar un tester. Estos se pueden agrupar en tres principales fases.

<u>Resolución de dudas</u> → Se reúnen el tester y el cliente. El cliente le proporciona los activos que necesita el tester y quiere saber cuándo terminará el *testing* y cuánto le costará. El tester necesita que se le resuelvan todas las dudas para poder empezar el *testing* y para comprender completamente el servicio, así como proporcionar el coste y el tiempo que le llevará testearlo.

Realización de activos → El tester tiene todos los activos necesarios para realizar el *testing*. Según se van preparando, el tester es capaz de visualizar con más detalle la ejecución que tendrá que realizar y podrá estimar con más exactitud el tiempo que le llevará.

Ejecución y entrega → El tester ejecuta las pruebas, recopila los resultados y le entrega al cliente todo. Una vez entregado, el cliente deberá decidir según los resultados si dar por terminado el servicio o si desea abrir un nuevo proceso en el caso de que el servicio haya de ser modificado.

4.2 Documento técnico de alto nivel (DTAN)

Una vez que el tester ha acordado todos los términos con el cliente y éste le ha resuelto todas las dudas, será capaz de empezar el proceso del *testing*.

Todo *testing* comienza por el documento más básico y esencial, documento proporcionado por el cliente y que el tester habrá de tenerlo a mano durante todo el proceso del *testing*, el DTAN.

El DTAN es un documento que engloba todos los aspectos y características del servicio. Antes de comenzar el proceso del *testing*, el tester ha de asegurarse que todos los datos que necesita se encuentra en el documento y no existe incongruencias ni contradicciones (es decir, el tester ha de validar el DTAN y revisarlo de forma minuciosa para saber que todo es correcto).

Antes de comenzar la fase de validación el tester ha de saber el framework que utiliza el servicio y en qué fase se encuentra la petición/servicio (la tecnología del framework indicará el proceso del *testing* que el tester deberá seguir).

El DTAN ha de tener un historial de cambios que indique en qué fase se encuentra el servicio, en este historial se puede observar una de las siguientes fases:

- Nuevo desarrollo
    - Si el servicio es un nuevo desarrollo, significa que nadie ha validado el DTAN y, por lo tanto, ha de validarse completamente.
- Evolutivo
    - Si el servicio es un evolutivo, significa que parte de la funcionalidad ya está validada y por lo tanto solo habrá que revisar únicamente las nuevas operaciones del servicio.
- Gestión del cambio
    - Si el servicio es una gestión del cambio significa que se han producido cambios en las operaciones/funcionalidades del servicio, en esta fase, hay que validar si los cambios realizados en el servicio son correctos y viables.
- Correcciones
    - Finalmente, si el servicio está en esta fase, el tester además de validar el DTAN ha de analizarlo para saber si va a ser necesario testear el servicio o si los cambios no son tan relevantes como para influir de alguna forma en el resultado del *testing*, y, por lo tanto, no sería necesario realizar dicho proceso.

4.2.1 Fase de validación

Una vez tenga claro el tester en qué fase se encuentra el servicio, el tester empezará a validar el DTAN. A continuación, se indican los apartados más importantes del documento y en qué se ha de centrar el tester.

Cada servicio lo conforman un conjunto de operaciones donde cada operación tendrá su propia funcionalidad y, por lo tanto, en esta fase el tester ha de asegurarse de validar todas y cada una de las operaciones por separado. Además, en la fase de pruebas funcionales el tester tendrá que ejecutarlas una a una.

Operaciones del servicio

El DTAN indica las operaciones del servicio, especificando el objetivo de cada operación y un breve resumen sobre estas.

Cada operación tiene un conjunto de parámetros de entrada (cada parámetro corresponde con una columna de una tabla donde se almacena su valor) que recibirá de parte del usuario o de otro servicio (la salida de un servicio puede ser la entrada de otro), además de un conjunto de parámetros de salida (por cada petición hay una respuesta).

- Parámetros entrada/salida (véase un ejemplo en *la figura 11*)
    - El tester ha de saber si, para cada parámetro de entrada del servicio, tanto la columna a la que está asociada al parámetro como la tabla a la que está asociada la columna existen y tienen el mismo tipo. Además, la longitud del parámetro ha de ser viable con el de la columna de la tabla.
    - En este apartado, el tester puede ir más lejos en la validación y tratar de ver la finalidad de cada parámetro para asegurarse de que éste es necesario y tiene un tipo y longitud acorde con su finalidad (este paso no mejora la calidad del *testing*, pero si la del producto, esto el cliente también lo valora).

Parámetros de entrada						
Nombre	**Descripción**	**Tipo dato**	**Obligatorio**	**Valor defecto**	**Formato**	**Rango/ Valores posibles**
localeLanguage	I18N	String	S	[es]	N/A	N/A
localeCountry	I18N	String	S	[ES]	N/A	N/A
localeVariant	I18N	String	S	N/A	N/A	N/A
Lista de Datos		List	N	N/A	N/A	[0…4]
codProveedor	Código identificativo de tercero	String	N	N/A	N/A	[15]
codProducto	Código identificativo de la localización	Long	N	N/A	N/A	[0…999999999999]
fechaEntrega	Grupo de compra o pedido externo	Date	N	N/A	N/A	[YYYY-MM-DD]
fechaCaducidad	Descripción corta del tipo de localización	Date	N	N/A	N/A	[YYYY-MM-DD]

Parámetros de salida				
Nombre	**Descripción**	**Tipo de dato**	**Formato**	**Rango**
codError	Código del error	String	N/A	N/A
desError	Descripción del error	String	N/A	N/A

Figura 11. *Parámetros de entrada y salida de un servicio.*

Una vez el tester ha validado todos los datos funcionales de la operación, habrá de validar los datos de rendimiento que, como en la fase funcional, el tester tendrá que rendirla operación por operación.

- Cuadro de rendimiento (se muestra un ejemplo en *la figura 12*)
    - En este apartado el tester ha de asegurarse que el número de hilos concurrentes (es decir, peticiones/usuarios que acceden y usan el servicio al mismo tiempo) es permitido por el entorno (cada servicio se ejecuta en un entorno que es capaz de recibir un número limitado de peticiones de forma simultánea) en el que se va a ejecutar el servicio y, en el caso de que use la base de datos, que el número de registros máximos que contempla ese servicio es viable con el tamaño de las tablas que usará (en otras palabras, si un servicio registra a un usuario y el servicio contempla el registro de 1000 usuarios, el tester ha de asegurarse de que la tabla es capaz de almacenar 1000 registros para que pueda ejecutar la prueba de rendimiento).

Consideraciones para Rendimiento			
CONFIGURACIÓN DE LA PRUEBA	Usuarios concurrentes	Número medio	5
		Número máximo	12
	Frecuencia de uso		50 veces al día
	Escenario de pruebas		☐ E1 - Acciones que realizan todos los Usuarios disponiendo de menos de 4 horas. Por ejemplo: pedidos que deben realizarse entre las 8:00 y las 9:00 ☐ E2 - Acciones que realizan todos los Usuarios disponiendo entre 4 y 12 horas al día ☒ E3 - Acciones que realizan todos los Usuarios varias veces a lo largo de la semana *Acciones = invocaciones por operación*
	Excepciones al estándar para la prueba de Carga		No aplica
CAMINO LÓGICO A RENDIR	Especificación del comportamiento		Camino lógico determinado en REQ-FUN-0001
	Parámetros de entrada no obligatorios a informar		No aplica
OTROS	Tamaño de la listas		Lista de datos: 0..4
	Registros por página		No aplica
	¿Datos en preproducción?		- *No*

Figura 12. *Ejemplo de un cuadro de rendimiento con sus correspondientes datos.*

Una vez validados todos los datos, el tester puede empezar a vislumbrar el camino que tomará la operación para tener una idea de qué es lo que pretende y la razón por la que es necesaria.

- Comportamiento (se muestra un ejemplo en la *figura 13*)
    - En el DTAN ha de aparecer el comportamiento del servicio, tanto el principal como el secundario, además, debe indicar los errores que gestionar según los posibles sucesos durante la ejecución (valores de los parámetros de entrada inadecuados, tratar de insertar un registro sin su clave ajena, etc...).

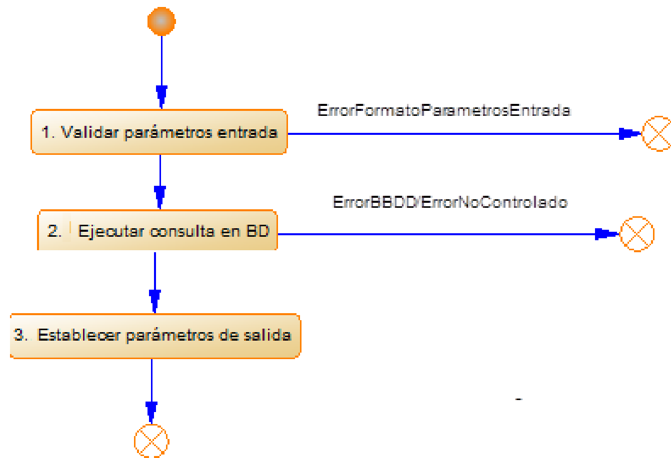


Figura 13. *Comportamiento de una operación del servicio*

Por último, si el tester ha validado toda la parte funcional y el cuadro de rendimiento de la operación, solo ha de asegurase que los errores engloban todos los posibles fallos que pueden darse durante la ejecución de la operación.

- Errores (véase un ejemplo en *la figura 14*)
    - El DTAN tendrá que indicar los errores que gestiona el servicio. En cada uno de ellos tendrá que ser preciso, no solo de la salida del servicio cuando estos se produzcan, si no de la traza que éste generará. Esto es muy importante para que el servicio sea lo más transparente posible hacia el usuario y para que el tester pueda identificar mejor los posibles defectos del servicio (si los tiene).

1.2 Errores gestionados

1.2.1 ErrorFormatoParametrosEntrada

Nombre:	ErrorFormatoParametrosEntrada
Descripción:	Alguno de los parámetros de entrada no cumple su formato
CodigoError:	Aplicacion.error.formatoParametrosEntrada
Mensaje Amigable:	Parámetros de entrada con formato incorrecto. [Parametros: {0}] 1. {0}: Parámetros (separado por comas) de todos los parámetros de entrada que no cumplen su formato
Mensaje Trazado:	{0} – {1} – {2} 1. {0}: Nombre del servicio web. 2. {1}: Nombre de la operación del servicio web. 3. {2}: Mensaje descrito en el apartado Mensaje Amigable.

Figura 14. *Ejemplo de error gestionado por un servicio*

4.3 Documento técnico detallado (DTD)

Si todo el DTAN es correcto y no tiene ningún fallo, el tester ya tiene una visión del servicio y una idea clara del *testing* que ha de hacer. Por lo tanto, podrá pasar al DTD, un documento más específico y concreto de lo que hace el servicio. En el caso de que el tester haya descubierto algún fallo del DTAN, no podrá continuar hasta que el cliente corrija todos los fallos y el tester lo valide por completo.

Este documento, al igual que el DTAN, debe proporcionártelo el cliente y su contenido es el conjunto de clases y métodos que usa el servicio, explicando en cada método de cada clase:

- La descripción del método, los parámetros de entrada y salida, el comportamiento del método y las llamadas a otros métodos.

Este documento está formado por clases y métodos escritos en un lenguaje de programación, y dado que nunca es fácil leer el código escrito por otra persona, ayuda en gran medida haber revisado antes el DTAN y tenerlo cerca durante la validacion del DTD (como se ha mencionado en la introducción, el proceso sigue un estándar que no se puede alterar ya que el orden existe por razones como esta).

4.3.1 Validación del DTD

El tester ha de ver que los parámetros de entrada y salida de cada método coincide con el DTAN, que la descripción es adecuada con su comportamiento y que los comportamientos de los métodos son acordes al comportamiento que indica el DTAN.

En este punto, el tester no sólo ha de saber que los datos de la operación son correctos, sino qué es lo que hacen y, sobretodo, en qué momento se usan (algo muy útil si se producen errores durante la ejecución).

- En el caso de que salte un error mientras el tester está ejecutando un caso funcional, si el tester conoce los métodos que ejecuta y qué datos usa cada método, será capaz de descubrir la razón por la que no ha funcionado.

4.4 Fase de pruebas funcionales

Una vez validado el DTAN y el DTD, se ha de comenzar con las pruebas funcionales para verificar que el servicio funciona como se espera y no tiene ningún defecto ni error en su desarrollo.

En esta fase se trata: el activo que deberá seguir el tester en todo momento para testear la funcionalidad del servicio de forma completa(PLP), los activos con los valores de los parámetros de entrada junto con los registros de las tablas necesarias para cada caso (ETF y DAT) y el WSDL junto con el script funcional(SCF) que forman el proyecto que ha de usar el tester para testear la funcionalidad del servicio (Activos ya explicados brevemente en el apartado 4).

4.4.1 Plan de pruebas (PLP)

El primer activo que el tester tendrá que empezar a realizar es el PLP, este documento reunirá todas las pruebas funcionales que el tester ejecutará y será el informe más importante para el cliente. Si algo falla en este activo, el cliente no sólo no tendrá un software de calidad, sino que tampoco tendrá un servicio plenamente funcional.

Para este activo solo se requiere la herramienta microsoft excel [8], a pesar de que puede parecer una herramienta simple, es una de la más completas y en la mayoría de empresas *testing* va a ser obligatorio que el tester sepa utilizarla para aprovechar al máximo su potencial. Si lo hace, optimizará el proceso de este activo de forma considerable.

El PLP es un documento Excel que recopila una serie de casos funcionales para cada operación. Cada caso tendrá un conjunto de valores distintos que probará una validación de entrada, un error del servicio o una de todas las posibles salidas tanto del comportamiento principal como del secundario.

Al final de la realización del activo, cada operación debe tener los suficientes casos para verificar todas las validaciones de los parámetros de entrada/salida, todos los caminos y salidas del comportamiento de la operación y todos los errores controlados. Además de un error no controlado (esto es debido a que en ningún momento el usuario que utilice el servicio debe ver un error sin ser gestionado y, por lo tanto, sin tener una salida amigable).

Cuando se prepare la ejecución, existe una técnica que se denomina “DataSource”, esto ayuda al tester a hacer lo que se denomina comúnmente el “caso escalera”. Consiste en poner todos los parámetros optativos a nulo e ir informando uno de ellos en cada registro (habrá tantos registros como parámetros optativos). Este caso permite al tester verificar que los parámetros pueden ser realmente nulos y el servicio no tiene ningún problema ni error con ninguno de los registros.

Por último, cada caso ha de tener una casilla donde el tester pueda indicar si el resultado del *testing* para ese caso es “OK” o “KO”, esto es muy útil cuando se realiza la entrega (así el cliente sabe qué funcionalidad del servicio falla y el desarrollador puede localizar más fácilmente cual es el motivo del fallo) además de cuando la petición está en la fase correctiva, debido a que el tester puede saber qué casos fallaron en el *testing* que se realizó anteriormente y, si es posible, testear solo esos casos.

4.4.1.1 Ejemplo explicativo

Para tener una visión más clara del activo, en este apartado se muestra un ejemplo del activo PLP. Hay que tener en cuenta, que el documento no ha de ser muy complicado ya que el cliente ha de saber con un simple vistazo cómo han ido las pruebas funcionales.

Operación de Consulta

		ENTRADA	SALIDA			EJECUCIÓN	
TC ID	Descripcion	Base	Resultado esperado	Base de salida	Comprobaciones adicionales	Resultado obtenido	Comentarios
OP_01-F-001	Se consulta un único registro.	OP_01-BaseEntrada-01	OK	OP_01-BaseSalida-01	Comprobar que se devuelve un registro		
OP_01-F-002	Se consultan dos o más registros.	OP_01-BaseEntrada-02	OK	OP_01-BaseSalida-02	Comprobar que se devuelve al menos dos registro		
OP_01-F-003	Se consultan dos o más registros.	OP_01-BaseEntrada-03	OK	OP_01-BaseSalida-03	Comprobar que se devuelve al menos dos registro		
OP_01-F-004	Se consulta un registro con todos los parametros informados	OP_01-BaseEntrada-04	OK	OP_01-BaseSalida-04	Comprobar que se devuelve un registro		
OP_01-F-005	Se consulta un registro con caracteres especiales	OP_01-BaseEntrada-05	OK	OP_01-BaseSalida-05	Comprobar que se devuelve un registro		
OP_01-F-006	Se consulta un registro con parametros con cadena vacia	OP_01-BaseEntrada-06	OK	OP_01-BaseSalida-06	Comprobar que se devuelve un registro		
OP_01-F-007	Se consulta un registro con parametros con espacios en blanco	OP_01-BaseEntrada-07	OK	OP_01-BaseSalida-07	Comprobar que se devuelve un registro		
OP_01-F-008	Se hace una consulta con parametros con formato incorrecto	OP_01-BaseEntrada-08	KO	OP_01-BaseSalida-08	Comprobar que devuelve errorFormatoParametros		
OP_01-F-009	Se consulta un registro que no existe	OP_01-BaseEntrada-09	KO	OP_01-BaseSalida-09	Comprobar que devuelve errorBBDD		
OP_01-F-010	Se consulta un registro que no existe	OP_01-BaseEntrada-10	KO	OP_01-BaseSalida-10	Comprobar que devuelve errorNoControlado		

Figura 15. *Ejemplo de un PLP con sus casos funcionales correspondientes*

En la *figura 15* se puede observar un ejemplo de cómo ha de listarse los casos funcionales para cada operación con algunos apartados agregados. A cada caso se le ha agregado: una descripción que indica el objetivo de éste, una sección que hace hincapié en la comprobación más importante y una sección de comentarios en el apartado de la ejecución por si el resultado es un "KO" y el tester ha encontrado el origen de ese fallo.

4.4.1.2 Fase de validación

Como en los anteriores documentos, debe haber un proceso de validación donde el tester, además de asegurarse de que el activo está completo, ha de pensar desde el punto de vista del usuario en el momento de escribir los datos de entrada para cada caso (estos pueden ser incorrectos o una serie de caracteres especiales que el servicio ha de ser capaz de gestionar de una forma u otra).

En este documento el tester ha de validar:

- <u>Nombres y estructura del documento</u>
    - El tester antes de entrar en materia, ha de verificar que el documento tiene todas las operaciones y que, para cada una de ellas, tiene un caso de salida por cada caso de entrada. Finalmente, una vez verificado que está completo, ha de fijarse que los nombres de las operaciones y de los parámetros son correctos.

- <u>Parámetros de entrada</u>
    - El tester ha de verificar que el servicio contempla todas las posibles entradas de los parámetros, esto incluye:
        - Con respecto a los parámetros de tipo **string/varchar** → recibir una cadena vacía, espacios en blanco o no recibir ningún valor o un valor nulo. Según el DTAN, en estos casos debería acabar con un error controlado debido al formato de un parámetro de entrada o con un valor por defecto. De no ser así, habría que informar este defecto para que se subsanase.
        - Con respecto a los parámetros de tipo **integer, long y double** → recibir una longitud mayor que la que permite la tabla a la que va a ser insertado o un rango de valor inadecuado (-1).
        - Con respecto a las **listas** → en este caso, el tester ha de comprobar el comportamiento del servicio cuando sólo se le informan los parámetros obligatorios de la lista, cuando no se informa la lista completa o cuando ésta tiene varios registros con distintas combinaciones de valores entre sus parámetros (este caso se usa para comprobar que el servicio es capaz de manejar listas con varios registros y confirmar si se aplica la ordenación o paginación del servicio).

Operación de Consulta

				Lista de Datos			
	localeLanguage (obligatorio)	localeCountry (obligatorio)	localeVariant (obligatorio)	codProveedor (optativo)	codProducto (optativo)	fechaEntrega (optativo)	fechaCaducidad (optativo)
	* String	* String	String	* Varchar (15)	* Long (12)	* Date	* Date
OP_01-BaseEntrada-01	es	ES	V	null	null	null	null
OP_01-BaseEntrada-02	es	ES	V	Proveedor 2	null	null	null
				null	1000000002	null	null
				null	null	sysdate	null
				null	null	null	2017-06-10
OP_01-BaseEntrada-03	es	ES	V	[DataSource]	[DataSource]	[DataSource]	[DataSource]
OP_01-BaseEntrada-04	es	ES	V	Proveedor 1	1000000001	sysdate	sysdate+1
OP_01-BaseEntrada-05	es	ES	V	*_/>Carc.#,%$_*	123456789012	2017-01-12	sysdate
OP_01-BaseEntrada-06	es	ES	V	[cadena vacia]	[cadena vacia]	null	null
OP_01-BaseEntrada-07	es	ES	V	[espacios en blanco]	[espacios en blanco]	null	null
OP_01-BaseEntrada-08	es	ES	V	abcdefghijklmnop	-1	2017-01-12T12:10:02	2017-01-15T20:10:02
OP_01-BaseEntrada-09	es	ES	V	abcdefghijklmno	999999999999	null	null
OP_01-BaseEntrada-10	es	ES	SELECT	Proveedor 1	1000000001	sysdate	sysdate+1

DataSource			
Proveedor 2	null	null	null
null	1000000002	null	null
null	null	sysdate	null
null	null	null	2017-06-10

Figura 16. *Ejemplo de entrada del PLP*

En la *figura 16* se puede ver la entrada de diversos casos, de los cuales hay que destacar:

- el caso 2 y 3 son iguales salvo por el hecho que el caso 3 usa la técnica DataSource (en la parte inferior de la *figura 16* está el cuadro con los valores que debe de tener el datasource).

- el caso 5 prueba la validación sobre los caracteres especiales.
- el caso 8 prueba la validación de las fechas. Con este caso se fuerza que la fecha tenga horas y se comprueba si el campo y el servicio permite ser tan específico con la fecha o no. Además, el tester puede descubrir con este caso, si el servicio trunca las fechas o las acepta completamente (por este motivo, siempre debe de haber un caso que tenga una fecha específica en el caso de que exista un parámetro de tipo *date*).
- Finalmente, además de los casos de cadena vacía o espacios en blanco, en el último caso se ha puesto en uno de los parámetros obligatorios especiales, (los parámetros especiales son aquellos que solo permiten unos pocos valores muy concretos) "localeVariant", un valor incorrecto para que el servicio lance el error "errorNoControlado" (de esta forma, el tester puede confirmar que los errores no controlados también los gestiona el servicio).

- Parámetros de salida
  - El tester ha de ver todas las posibles salidas que le permite el servicio, tanto los caminos que acaban en error como todas las posibles bifurcaciones que puede haber dentro del comportamiento del servicio.

Operación de Consulta

	codError	desError
OP_01-BaseSalida-01	null	null
OP_01-BaseSalida-02	null	null
OP_01-BaseSalida-03	null	null
OP_01-BaseSalida-04	null	null
OP_01-BaseSalida-05	null	null
OP_01-BaseSalida-06	null	null
OP_01-BaseSalida-07	null	null
OP_01-BaseSalida-08	logis.common.error.formatoParametrosEntrada	Parámetros de entrada con formato incorrecto. [Parametros: codProveedor,codProducto,fechaEntrega,fechaCaducidad]
OP_01-BaseSalida-09	logis.common.error.BBDD	Error ejecutando una operación sobre base de datos.
OP_01-BaseSalida-10	logis.common.error.noControlado	Error en el servicio

Figura 17. *Ejemplo de salida del activo PLP*

Este ejemplo de salida (*Figura 17*) muestra los dos parámetros que todo servicio debe tener (como mínimo), un parámetro que indique el código del error y un parámetro que dé una descripción de éste. En este ejemplo se puede ver: los siete casos funcionales cuya salida debería ser con los dos parámetros a "null", un caso que lanza un error por algún parámetro de entrada incorrecto, un caso por algún fallo con la base de datos y, finalmente, el caso que lanza un error que no controla el servicio (pero que si debe ser capaz de gestionar).

Este último caso debe tener un parámetro con algún valor extraño para que el servicio tenga una salida para aquellos casos que no controla, este caso se realiza para verificar que tiene una salida "amigable" y una traza del error hasta para los errores que el servicio no contempla.

4.4.2 Entrada funcional (ETF) [FW2, FW3]

Una vez realizado y validado el PLP, el tester podrá realizar este documento (de no ser así, no podrá continuar hasta que el PLP no sea perfecto). Este activo se realiza de diversas formas según el tipo de framework (online o local) debido a los tipos de entrada que pueden recibir (en framework local puede recibir un fichero en lugar de un valor de una variable), a pesar de que lleva tiempo realizar este documento, su valor y utilidad a largo plazo lo compensa (esto se debe a que ahorra mucho tiempo al tester si la petición/servicio se encuentra en una fase que no es “nuevo desarrollo”, estas fases están definidas en el apartado 4.2 y se explica el ahorro de tiempo a continuación y en el apartado 4.4.7).

Este fichero ha de tener una inserción por cada caso funcional del PLP y cada una de las inserciones tendrá los valores de ese caso (cada valor corresponderá a cada uno de los parámetros). De esta forma, se pretende tener una tabla con tantos registros como casos del PLP donde cada registro tendrá la entrada del caso del PLP al que corresponde.

Para este documento existe una herramienta privada denominada “*ETF Generator*”. Esta herramienta te permite generar el activo ETF automáticamente a partir del PLP, aunque obliga a seguir una plantilla muy precisa para que sea capaz de leerlo. Esta herramienta básicamente es un script que permite al tester extraer los datos del excel (concretamente, los datos de la hoja de entrada del PLP) y ponerlos en formato SQL (es decir en una instrucción “*insert*” teniendo los valores de los parámetros separados por comas).

Este activo tiene una importancia muy reciente en el proceso del *testing*; El objetivo de este fichero en el proceso es agilizar la asignación de valores a cada parámetro de la operación. Para los servicios en la fase de nuevo desarrollo, facilita la asignación de valores, pero su verdadero potencial es cuando el servicio está en cualquiera de las otras fases y permite cambiar los valores de una forma notablemente más rápido.

A continuación, se muestra un ejemplo para clarificar la explicación del activo. El siguiente ejemplo ha sido creado especialmente para el trabajo con el único fin de ayudar al lector a comprender el activo y todo lo explicado anteriormente.

4.4.2.1 Ejemplo explicativo

Siguiendo con el ejemplo del PLP, se puede ver en la *figura 18* la creación de una tabla específica para la operación del servicio. Posteriormente, se encuentran el conjunto de inserciones que se tendrán que hacer, cada una con los valores del caso que le corresponden (según el PLP).

```sql
------------------------------------
-- OP_01 -> Operacion de Consulta --
------------------------------------

-------------------------
-- Borrado de datos
-------------------------
DELETE FROM APP_ServicioCompleto_OP01;
COMMIT;
-------------------------
-- Borrado de tabla
-------------------------
DROP TABLE APP_ServicioCompleto_OP01;
COMMIT;
-------------------------
-- Creación de la tabla
-------------------------
CREATE TABLE APP_ServicioCompleto_OP01 (
    TC_ID VARCHAR2(200 BYTE) NOT NULL ENABLE,
    LOCALELANGUAGE VARCHAR2(2000 BYTE),
    LOCALECOUNTRY VARCHAR2(2000 BYTE),
    LOCALEVARIANT VARCHAR2(2000 BYTE),
    CODPROVEEDOR VARCHAR2(2000 BYTE),
    CODPRODUCTO VARCHAR2(2000 BYTE),
    FECHAENTREGA VARCHAR2(2000 BYTE),
    FECHACADUCIDAD VARCHAR2(2000 BYTE) DEFAULT '[ver PLP]'
);
COMMIT;

-------------------------
-- Inserción de datos
-------------------------

-- OP_01-BaseEntrada-01
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-001', 'es', 'ES', 'V', null, null, null, null, default);
-- OP_01-BaseEntrada-02
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-002', 'es', 'ES', 'V', 'Proveedor 2', null, null, null, default);
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-002', 'es', 'ES', 'V', null, 1000000002, null, null, default);
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-002', 'es', 'ES', 'V', null, null, sysdate, null, default);
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-002', 'es', 'ES', 'V', null, null, null, TO_DATE('2017-06-10','YYYY-MM-DD'), default);
-- OP_01-BaseEntrada-03
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-003', '[no informado]', 'ES', 'V', null, null, null, null, default);
-- OP_01-BaseEntrada-04
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-004', 'es', 'ES', 'V', 'Proveedor 1', 1000000001, SYSDATE, SYSDATE+1, default);
-- OP_01-BaseEntrada-05
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-005', 'es', 'ES', 'V', '*_/>Carc.#,%$_*', 123456789012, TO_DATE('2017-01-12','YYYY-MM-DD'), sysdate, default);
-- OP_01-BaseEntrada-06
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-006', 'es', 'ES', 'V', '', '', null, null, default);
-- OP_01-BaseEntrada-07
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-007', 'es', 'ES', 'V', '[espacios en blanco]', '[espacios en blanco]', null, null, default);
-- OP_01-BaseEntrada-08
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-008', 'es', 'ES', 'V', 'abcdefghijklmnop',-1,'2017-01-12T12:10:02','2017-01-15T20:10:02', default);
-- OP_01-BaseEntrada-09
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-009', 'es', 'ES', 'V', 'abcdefghijklmno',999999999999, null, null, default);
-- OP_01-BaseEntrada-10
INSERT INTO APP_ServicioCompleto_OP01 VALUES ('OP_01-F-010', 'es', 'ES', 'SELECT', 'Proveedor 1', 1000000001, SYSDATE, SYSDATE+1, default);
```

Figura 18. *Ejemplo del activo ETF con sus correspondientes datos y estructura*

Para finalizar, arriba de la tabla se puede visualizar dos instrucciones que permiten eliminar la tabla o borrarla. Esto es útil cuando el servicio se encuentra en un estado posterior al nuevo desarrollo y se han hecho cambios que deben aplicarse a la tabla.

4.4.3 Entrada funcional (ETF) [Standalone]

A diferencia de los servicios online, cuando el servicio usa framework Standalone ha de tener dos ficheros ETF (uno para la entrada y otro para la salida) por cada operación. Estos activos deberán guardarse con la extensión ".csv" compuesto por al menos 2 columnas (este es el formato aprobado por la factoría y se usa debido a la explicación que hay a continuación y la utilización del ETF en el script funcional, véase en el apartado 4.4.9):

- En el caso del ETF de entrada: la primera columna hace referencia al identificador del caso, la segunda columna y las siguientes corresponderán con los parámetros, es decir, habrá tantas columnas como parámetros tenga el servicio, además de la columna que identifica la línea/caso (esta columna servirá durante la ejecución para identificar los valores de un caso de otro, esto se muestra en mayor detalle en el apartado 4.4.9).

4.4.3.1 Ejemplo explicativo

En la *figura 19* se puede observar el formato que ha de tener un ETF de un servicio Standalone. Este activo no tiene ninguna complejidad sintáctica y lo que más hay que destacar es que el nombre de todos los parámetros han de ser exactos y no debe haber ninguna columna de más (esto se consigue abriendo el activo con un editor de texto y eliminar manualmente la última línea en blanco que crea el excel por defecto).

numTC	localeLanguage	localeCountry	localeVariant	codProveedor	codProducto	fechaEntrega	fechaCaducidad
OP_01-BaseEntrada-01	es	ES	V	null	null	null	null
OP_01-BaseEntrada-02	es	ES	V	Proveedor 2	null	null	null
OP_01-BaseEntrada-02	es	ES	V	null	1000000002	null	null
OP_01-BaseEntrada-02	es	ES	V	null	null	sysdate	null
OP_01-BaseEntrada-02	es	ES	V	null	null	null	42896
OP_01-BaseEntrada-03	es	ES	V	Proveedor 2	null	null	null
OP_01-BaseEntrada-03	es	ES	V	null	1000000002	null	null
OP_01-BaseEntrada-03	es	ES	V	null	null	sysdate	null
OP_01-BaseEntrada-03	es	ES	V	null	null	null	42896
OP_01-BaseEntrada-04	es	ES	V	Proveedor 1	1000000001	sysdate	sysdate+1
OP_01-BaseEntrada-05	es	ES	V	*_/>Carc.#,%$_*	123456789012	42747	sysdate
OP_01-BaseEntrada-06	es	ES	V	[caden vacia]	[caden vacia]	null	null
OP_01-BaseEntrada-07	es	ES	V	[espacios en blanco]	[espacios en blanco]	null	null
OP_01-BaseEntrada-08	es	ES	V	abcdefghijklmnop	-1	2017-01-12T12:10:02	2017-01-15T20:10:02
OP_01-BaseEntrada-09	es	ES	V	abcdefghijklmno	999999999999	null	null
OP_01-BaseEntrada-10	es	ES	SELECT	Proveedor 1	1000000001	sysdate	sysdate+1

Figura 19. *Ejemplo de ETF de entrada de un servicio con framework Standalone*

En el caso del ETF de salida, además de la columna que identifica los casos, tendrá que tener una columna que indique la salida (nulo si todo ha ido correctamente o la traza y el mensaje amigable si se trata de un caso de error).

numTC	codError	desError
OP_01-BaseSalida-01	null	null
OP_01-BaseSalida-02	null	null
OP_01-BaseSalida-03	null	null
OP_01-BaseSalida-04	null	null
OP_01-BaseSalida-05	null	null
OP_01-BaseSalida-06	null	null
OP_01-BaseSalida-07	null	null
OP_01-BaseSalida-08	logis.common.error.formatoParametrosEntrada	Parámetros de entrada con formato incorrecto. [Parametros: codProveedor,codProducto,fechaEntrega,fechaCaducidad]
OP_01-BaseSalida-09	logis.common.error.BBDD	Error ejecutando una operación sobre base de datos.
OP_01-BaseSalida-10	logis.common.error.noControlado	Error en el servicio

Figura 20. *Ejemplo de ETF de salida de un servicio Standalone*

En la *figura 20* se observa un ejemplo de ETF de salida donde hay que destacar: la columna que identifica cada caso (ya mencionada en el ETF de entrada) y los valores que han de tener los parámetros de salida según si se ha producido un error o si el servicio ha respondido a la petición con éxito.

4.4.4 Datos técnicos (DAT) [FW online, STD público]

Una vez que el tester sabe los casos que ha de ejecutar y la entrada que cada ejecución recibirá, ha de hacer el activo DAT. Un activo que afectará directamente a la base de datos y que, al igual que el ETF llena una tabla con los valores de los parámetros de entrada, el DAT se encarga de llenar las tablas de base de datos que necesite el servicio.

Para este activo se requiere de una herramienta capaz de interactuar con la base de datos, un ejemplo de herramienta puede ser "*Oracle SQLDeveloper*".

Este documento es necesario cuando la operación y, por lo tanto, el servicio, interactúa con la base de datos. De ser así, tendremos que tener en cuenta el tipo de operación realiza:

- En el caso de que sea una **operación que modifica o borra**: el tester tendrá que insertar los registros necesarios para que cuando el servicio trate de realizar la operación, el servicio encuentre al menos un registro y pueda ejecutarse con éxito.

- Si es una **operación de consulta**: el tester deberá hacer las inserciones adecuadas para que cuando se ejecute el servicio pueda encontrar los registros que cumplan las condiciones del "*where*".

- Si es una **operación de inserción**: se realizarán las inserciones pertinentes en el caso de que la tabla en la que el servicio insertará tenga claves ajenas. Por el contrario, si la tabla no tiene ninguna clave ajena, entonces no hay necesidad de realizar inserciones antes de ejecutar el servicio y, por lo tanto, este documento sería innecesario.

Independientemente de la operación que realiza el servicio, para realizar un buen *testing* es necesario que el DAT tenga lo que se denomina *registros basura*. Estos registros sirven para confirmar al tester que únicamente aplica la operación a los registros que debería, y el resto de registros de la tabla se quedan como están. Por lo tanto, cada caso tendrá al menos un registro que utilizará el servicio y al menos un registro basura que no se usará.

Este activo no se revisa y se asumirá que se ha realizado correctamente hasta que el tester no empiece la ejecución funcional. Si un caso falla y se localiza el error en el activo DAT, se corregirá y se continuará con la ejecución.

Se sigue este proceso por dos motivos:

- La mejor forma de estar seguro de que funciona es en el proceso de ejecución y, si ocurre algún fallo, el log ayuda enormemente a localizar el error. Luego, este error se puede arreglar en pocos minutos y no interrumpe apenas el proceso. (en estos casos el tester puede comprobar la importancia de las trazas de error, trazas que aparecen en el DTAN y se visualizan en el ETF de salida, apartados anteriores ya explicados).

- El segundo motivo es porque revisar un DAT puede llegar a ser muy complejo, esta complejidad depende en gran medida del número de tablas que use y el comportamiento que tenga el servicio.
  Como añadido también hay que mencionar que en las empresas no siempre es el mismo tester quien valida un DTAN y quien ejecuta las operaciones, esto hace que suponga un esfuerzo considerable que un tester que solo ha de ejecutar unas pruebas tenga que estudiarse de forma tan concienzuda el comportamiento del servicio para revisar el DAT sin ningún log ni traza que le indique si existe un error o no.

4.4.4.1 Ejemplo explicativo

La *figura 21* muestra el DAT del primer caso, un caso simplificado que requiere de pocas tablas, pero que sirve para ver el potencial del activo y sus complejidades.

```
-----------------
-- OP_01-F-001 --
-----------------

-- Tablas de claves ajenas

INSERT INTO TABLA_PROVEEDORES (COD_N_PROVEEDOR,TXT_PROVEEDOR)
VALUES('Proveedor 1', 'proveedor de bebidas');

INSERT INTO TABLA_PRODUCTO (COD_N_PRODUCTO,TXT_PRODUCTO)
VALUES(1000000002, 'Agua');


-- Tabla que consulta el servicio, si fuera una operacion de insercion esta ultima insercion lo haria el servicio.

INSERT INTO TABLA_FINAL (COD_N_PROVEEDOR,COD_N_PRODUCTO, FECHA_ENTREGA, FECHA_CADUCIDAD)
VALUES('Proveedor 1',1000000002,TO_DATE('2017-04-15','YYYY-MM-DD'),TO_DATE('2017-06-10','YYYY-MM-DD'));

COMMIT;
```

Figura 21. *Ejemplo de DAT con sus correspondientes tablas.*

Aspectos importantes de la *figura 21* que hay que destacar:

- En este caso a pesar de que el servicio solo use la última tabla, se puede observar que tiene un código de proveedor que ha de estar en la tabla "TABLA_PROVEEDORES" y un código de producto que ha de estar en la tabla "TABLA_PRODUCTO".
  Es importante destacar que la última instrucción *insert* no se haría si fuera una operación de inserción ya que ésta instrucción es la que realizará el servicio cuando se ejecute (en este tipo de operaciones, el tester puede asegurarse que el DAT es correcto ejecutando la última instrucción y luego borrando el registro creado de la tabla, lleva tiempo, pero hacerlo en los primeros casos puede corregir un error e impedir que el tester lo "arrastre" a lo largo de la realización del resto de casos).

- En la *figura 21* también se observa que, a pesar de que el servicio usa una sola tabla, debido a las claves ajenas de ésta, las tablas del DAT se han triplicado. Si el servicio usara varias tablas y sus tablas ajenas tuvieran otras claves ajenas, el documento sería tremendamente complejo (esta es la razón de su complejidad y la razón por la que revisarlo puede ser una tarea ardua y costosa).

4.4.5 Datos técnicos (DAT) [Batch, Standalone]

Una vez realizado el ETF de Standalone si es un servicio de este framework, el tester ha de realizar el activo que corresponde con los datos que el servicio pretende recoger (en el caso de servicios con framework locales, no tiene por qué recoger los datos de una base de datos, puede ser de algún fichero que tenga el sistema).

Los servicios locales también pueden necesitar un DAT distinto. Esto es debido a que el servicio historifique, mueva o elimine ficheros de un directorio. Por lo tanto, es posible que el tester tenga que preparar ficheros vacíos o con un contenido en concreto. Además de que tenga unos nombres específicos y el fichero se encuentre en un directorio concreto (todos estos datos deben de estar indicados en el comportamiento del servicio del DTAN, véase en la sección 4.2).

Este activo es más simple en este framework ya que, de ser necesario, son documentos vacíos (el servicio se encarga de escribir en estos ficheros) o con fechas e identificadores donde el servicio tendrá que: ordenarlos, separar datos (por ejemplo: separar la hora del año de una fecha) o extraer los datos de dentro del documento y almacenarlos en otro directorio.

4.4.5.1 Ejemplo explicativo (caso real)

En la *figura 22* se observa un servicio real que ha sido testeado e implantado en el mundo real con éxito.

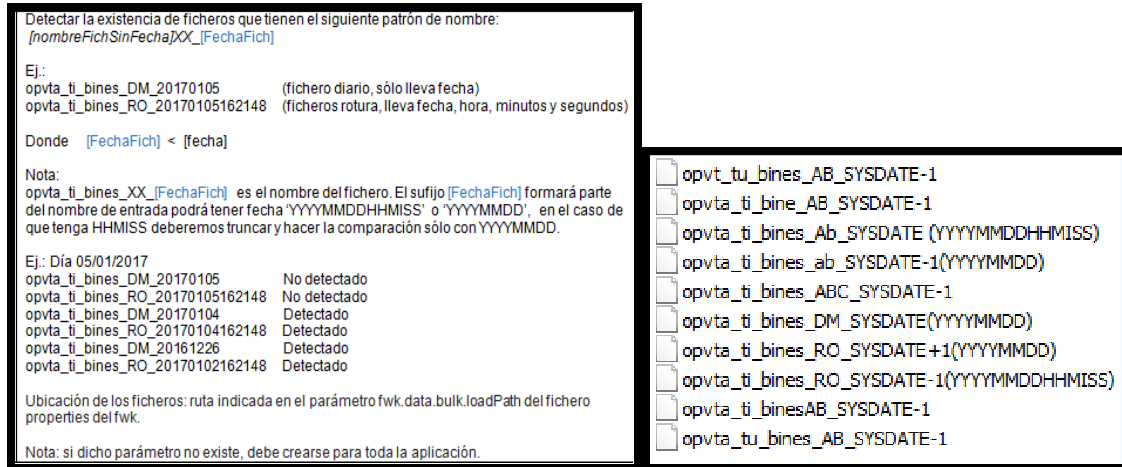


Figura 22. *Ejemplo de un DAT de un framework local*

En la parte izquierda se puede visualizar parte del comportamiento del servicio que explica de forma detallada como ha de ser el nombre de los ficheros, además de concretar cómo el servicio compara los ficheros (indicando que truncará las fechas y no tendrá en cuenta ni las horas, ni los minutos ni los segundos).

La parte derecha de la *figura 22* es un ejemplo de diversos ficheros que el servicio ha de comparar, en este ejemplo hay que destacar que existen ficheros con nombres que incumplen el formato indicado y, por lo tanto, el servicio no deberá tenerlos en cuenta durante la ejecución (al igual que en el apartado del DAT de servicios online se mencionaron *registros basura*, en este tipo de DAT deberá haber al menos un *fichero basura* para asegurar al tester que el servicio únicamente selecciona aquellos ficheros que cumplen con el formato indicado en el DTAN).

4.4.6 Activo WSDL [FW2, FW3, Standalone]

Una vez que el tester ya tiene los activos que necesita para los datos de entrada y los registros de la base de datos (o ficheros en el caso de ser un framework local que los tenga como entrada), el tester puede disponerse a realizar el activo (WSDL) que formará parte del proyecto con el que el tester realizará las pruebas funcionales del servicio.

El WSDL [10] es un fichero XML que se usa para crear un proyecto con la herramienta *SoapUI*, una de las herramientas más famosas que se usa en el *Testing*. Además de por ser una de las herramientas *Testing* más completas, las empresas pueden personalizarlo con documentos muy concretos que optimizan y mejoran el proceso (esta herramienta se ha explicado con algo más de detalle en el apartado 3).

Este documento debe tener todos los parámetros de entrada y de salida de las operaciones, así como indicar para cada uno de los parámetros su tipo e indicar si es obligatorio o no.

Con respecto a las fases del servicio, este activo tendrá que ser modificado en cualquier fase de gestión del cambio, evolutivo o correctivo siempre que afecte a los parámetros o se añada alguna operación (en caso de no ser así, este activo se mantiene inalterable). Por lo tanto, suele ser común que no se modifique el activo durante toda la vida útil del servicio.

Para terminar, una vez realizado el activo no se valida, ya que en cuanto se cree el proyecto *SoapUI*, se visualizará al instante las propiedades y parámetros del servicio, por lo que cualquier posible defecto, el tester lo descubrirá rápidamente antes de empezar siquiera con las pruebas. Además, cuando el tester vaya a crear un caso de una operación que se le haya olvidado en el wsdl o cuando vaya a introducir valores del PLP/ETF en un parámetro que no se encuentre en el WSDL, el tester lo sabrá en ese momento y en cuestión de segundos lo puede corregir y actualizar el proyecto *SoapUI* (otra de las razones por las que no se revisa es porque no penaliza de forma significativa en tiempo corregir este activo).

4.4.6.1 Ejemplo explicativo

En la *figura 23* se puede ver un ejemplo de WSDL creado específicamente para este proyecto, este documento se ha basado en el ejemplo del PLP (apartado 4.4.1).

```
<?xml version='1.0' encoding='ISO-8859-15'?><wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://
schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:ns1="http://schemas.xmlsoap.org/soap/http" name="
ED_ServicioCompleto">
    <wsdl:types>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://cxf.apache.org/arrays" attributeFormDefault="qualified"
elementFormDefault="qualified" targetNamespace="http://cxf.apache.org/arrays">
  <xsd:complexType name="Input_OperacionConsulta">
    <xsd:complexContent>
        <xsd:sequence>
          <xsd:element name="localeLanguage" type="xsd:string"/>
          <xsd:element name="localeCountry" type="xsd:string"/>
          <xsd:element name="localeVariant" type="xsd:string"/>
          <xsd:element name="codProveedor" nillable="true" type="xsd:string"/>
          <xsd:element name="codProducto" nillable="true" type="xsd:string"/>
          <xsd:element name="fechaEntrega" nillable="true" type="xsd:dateTime"/>
          <xsd:element name="fechaCaducidad" nillable="true" type="xsd:dateTime"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="Output_OperacionConsulta">
    <xsd:complexContent>
        <xsd:sequence>
          <xsd:element minOccurs="0" name="codError" nillable="true" type="xsd:string"/>
          <xsd:element minOccurs="0" name="desError" nillable="true" type="xsd:string"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Figura 23. *Ejemplo del activo WSDL con los parámetros de una operación*

En la *figura 23* se puede distinguir claramente los parámetros de entrada con los de salida (mediante el nombre del "complexType"), también hay que destacar que los parámetros optativos a diferencia de los obligatorios tienen la propiedad "nillable" (pueden ser nulos).

Estos parámetros optativos suele ser la parte que más complica al tester con relación a este activo debido a que no descubrirá un error de tipo "nullable" hasta que no salte el error **durante la ejecución** (hasta que el tester no ejecute el caso y el servicio le devuelva un error por recibir un parámetro obligatorio sin ningún tipo de valor asociado, no sabrá que hay un activo mal preparado y, con la traza del servicio, el tester le resultará fácil saber el motivo del fallo).

4.4.7 Script funcional (SCF) [FW2, FW3,STD público]

Cuando el tester ya posee todos los activos que necesita para el *testing* funcional, puede empezar a crear un proyecto *SoapUI* donde preparará y ejecutará los casos del PLP, usando los activos que ha ido creando según los vaya necesitando.

Lo primero que hay que hacer es abrir el *SoapUI* y crear un proyecto donde se crearán tantos conjuntos de casos como operaciones. Una vez creados se tendrá que crear cada caso del PLP y llegará el momento de usar todos los activos anteriormente realizados:

- El fichero WSDL: este activo permitirá poder crear casos según la operación (testSuit). Una vez creado el caso, el tester podrá visualizar tantos parámetros como tiene cada operación (y detectar algún fallo que pueda tener el WSDL en este aspecto).

- Activo ETF: Dentro del caso habrá que crear un "step" que con una consulta basándose en el nombre del caso y del conjunto en el que se encuentra, extraerá el registro correspondiente al activo ETF. De esta forma, se enlazará cada parámetro con su valor correspondiente (a continuación, se muestra y se explica de forma más detallada la explicación).

    - Ejemplo real (En los apartados que se necesite el activo WSDL se ha usado imágenes de servicios reales para la explicación, estos servicios se han implementado con éxito y se ha verificado su calidad)

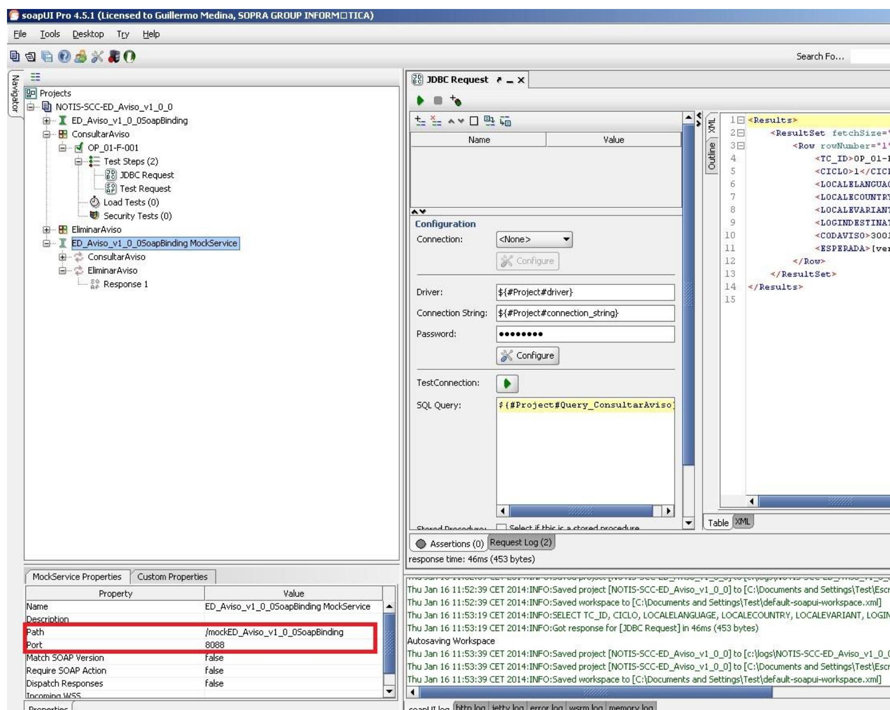


Figura 24. Visualización del SoapUI con la pestaña asociada al ETF

- En la *figura 24* se muestra un servicio con dos operaciones y, dentro de la primera operación, tiene dos “test”: el primer “test” corresponde al activo ETF (en la parte derecha de la *figura 24* se observa que se ha cargado el primer caso de la tabla con sus valores correspondientes). Finalmente, hay que destacar que este test tiene el apartado “SQL Query” que permite al tester escribir la consulta que recuperará la línea que nos interesa de la tabla auxiliar (tabla del activo ETF visto en el apartado 4.4.2).
    - Un ejemplo de Query en este caso sería: “select * from tabla_auxiliar where tc_id = testcase_name”

  Una vez que el “test” que corresponde al ETF está hecho, se generará el “test Request”. Un “test” que enlazará cada parámetro con una columna de éste y permitirá cargar los datos de entrada en el servicio. Devolviendo así, una salida del servicio y testeándolo (más adelante se usa un ejemplo del caso real para mejorar de una forma visual la explicación y facilitar la comprensión al lector).

- Ejemplo real (En los apartados que se necesite el activo WSDL se han usado imágenes de servicios reales para la explicación)

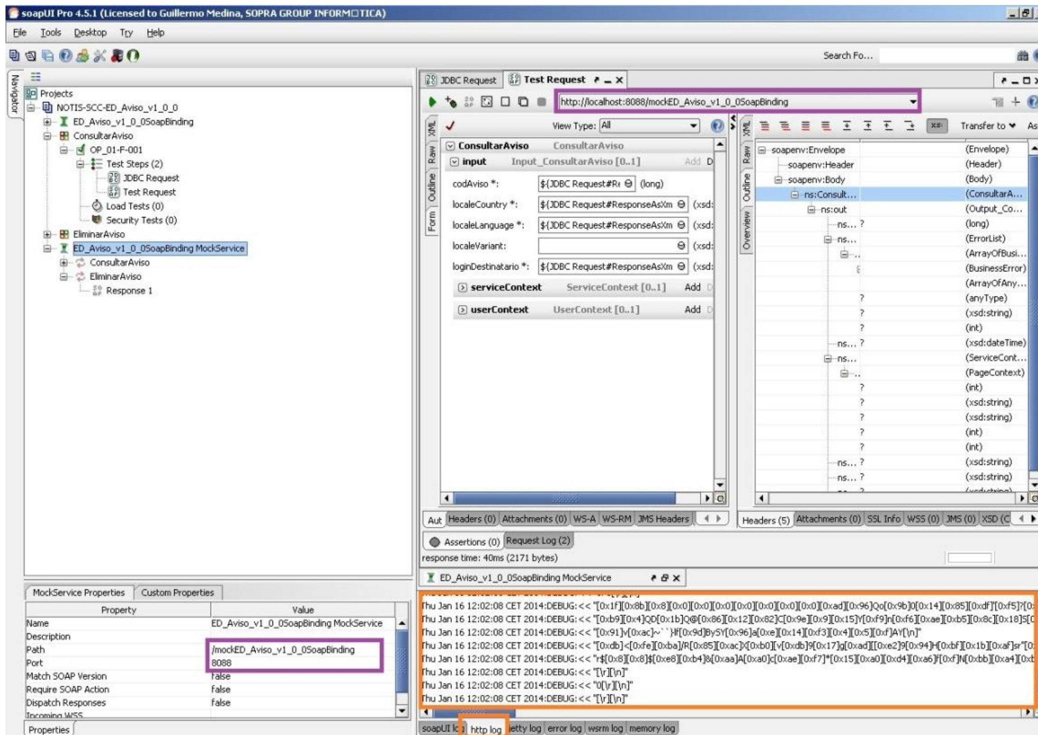


Figura 25. *Captura del SoapUI donde se muestra la pestaña "Test Request"*

- La *figura 25* es un ejemplo de "Test Request" donde cada parámetro está enlazado con el test del ETF (el test "JDBC Request que se ha mostrado anteriormente"), excepto el parámetro "localeVariant" (esto se debe a que su valor es nulo).
  En la parte derecha se puede visualizar una parte de la salida que ha devuelto el servicio a partir de la entrada introducida por el tester (se han ocultado los valores al ser un ejemplo privado). Por último, en la parte de arriba se observa con un cuadro morado el lugar donde hay que indicar el servidor en el que se encuentra el servicio a testear y en la parte inferior se encuentra las trazas del servicio y los mensajes de la herramienta *SoapUI* que lanza una ejecución.

- En este apartado se puede apreciar el potencial del activo ETF.
  Aunque se produjeran cambios o correcciones en el DTAN, el test JDBC seguiría teniendo la misma consulta y el test Request seguiría teniendo los mismos enlaces (salvo que se modificara el valor de un campo nulo). Por lo tanto, el SCF no se modificaría o prácticamente no lo haría y con actualizar el ETF sería suficiente.

- <u>Activo DAT</u>: Se ha de crear un tercer "step" (o insertar los datos manualmente con la herramienta *SQLDeveloper*) que tendrá las inserciones del activo DAT que corresponden con ese caso. Este "step" deberá estar antes del test "Test Request" para llenar la base de datos antes de realizar la llamada al servicio.

- Logs: Llegados a este punto, antes de ejecutar el caso funcional, queda hacer una de las cosas más importantes a la hora de testear un servicio, guardar los "logs" (son las trazas y el rastro que deja el servicio durante la ejecución que permite al tester saber qué ha hecho el servicio detallado paso a paso) que éste genera.
    - Esto es vital por tres motivos:
        - Sirve para corroborar y dejar constancia de que no sólo se ha realizado el *testing*, sino de cómo se ha realizado y qué respuesta ha tenido el servicio ante una entrada en concreto.
        - Permite al tester verificar tanto la salida de la base de datos como la traza y la salida del servicio.
        - Es muy útil cuando la salida no es la esperada y el tester ha de descubrir dónde está el defecto, para dejar constancia de una forma más rigurosa y localizada el error existente (en el caso de que no pueda solucionarlo el propio tester).
    - Para guardar los logs se usarán dos "steps" (con el mismo formato que el "step JDBC"):
        - Se ha de generar un "step" que permita al tester guardar tanto la salida del servicio como las trazas que éste genera en el directorio que se desee (*SoapUI* automáticamente guarda la salida del "test request").
        - Por último, se creará un paso comúnmente denominado SDB (saving database). Este paso permite guardar un log de la base de datos (a diferencia del anterior log, éste solo es necesario si el servicio modifica de alguna forma la base de datos).
            - Esto se consigue haciendo una consulta de la tabla sobre la que trabaja el servicio antes de ejecutar los demás "steps" y luego la misma consulta una vez ejecutado. De esta forma, el tester será capaz de ver los cambios que se han producido (este "step" también se puede realizar manualmente con la herramienta *SQLDeveloper*).

Una vez preparados todos los casos, sólo quedará indicarle al *SoapUI* en qué servidor está desplegado el servicio para empezar a ejecutar los casos sobre el entorno que previamente el tester le haya indicado (mencionado en el apartado 4.4.6).

- Un ejemplo del orden de ejecución es:
    - "Test" asociado al DAT para llenar la base de datos (de ser necesario).
    - "Test" SDB para guardar el estado de la base de datos antes de la ejecución.
    - "Test" asociado al ETF para cargar los valores del caso.
    - "Test" que ejecutará el servicio y enlazará cada valor cargado del ETF a cada parámetro del caso ("test Request").
    - "Test" SDB que guardará estado de la base de datos después de la ejecución.
    - "Test" que guardará el log de la ejecución con todas las trazas y mensajes que el servicio y la herramienta *SoapUI* generan.

Hay que dejar claro que, aunque hay algunos “test” que se pueden cambiar de orden durante la ejecución (por ejemplo, el “test” ETF puede ir al principio junto al DAT), hay que tener claro la funcionalidad de cada uno y saber que hay cierto orden inquebrantable (como que se deben ejecutar los “test” ETF y DAT antes de ejecutar el “Test Request” o que debe haber un “test” SDB antes y después de la llamada al servicio para guardar los cambios de la base de datos).

Cuando las pruebas funcionales hayan terminado, si el tester ha realizado todos los activos a la perfección y los ha ejecutado de forma correcta, podrá asegurar que funcionalmente el servicio es impecable y, por lo tanto, podrá pasar a la fase de seguridad.

4.4.8 Script funcional (SCF) [STD público]

Continuando con la explicación del anterior apartado, para ejecutar este activo con el framework standalone, el tester ha de tener en cuenta si las operaciones del servicio son públicas o privadas.

Si el servicio usa framework Standalone y tiene operaciones públicas, significa que hace uso de colas (una de las razones por las que se hace el proyecto mediante *SoapUI*, ya que permite usar la herramienta *HermesJMS* [11] muy fácilmente y con total compatibilidad.

HermesJMS es una herramienta que permite observar todas las colas del servicio, mostrando lo que hace el servicio y el impacto que tiene éste sobre las colas (en otras palabras, el tester puede visualizar los mensajes que encola el servicio durante la ejecución).

El tester solo tiene que indicar al *HermesJMS* el entorno donde se va a ejecutar. En ese momento, será capaz de visualizar todas las colas que éste contiene. Cuando ejecute un caso (como esta indicado en el apartado anterior), el tester solo habrá de guardar la salida de las colas que use el servicio en lugar de guardar la salida del *SoapUI* (como siempre, el tester ha de guardar logs y pruebas de su ejecución, tanto para que el cliente sepa que se ha probado como para ayudar al tester en futuras ejecuciones del servicio, si las tiene).

- Ejemplo Real (En los apartados que se necesite el activo WSDL se ha usado imágenes de servicios reales para la explicación)

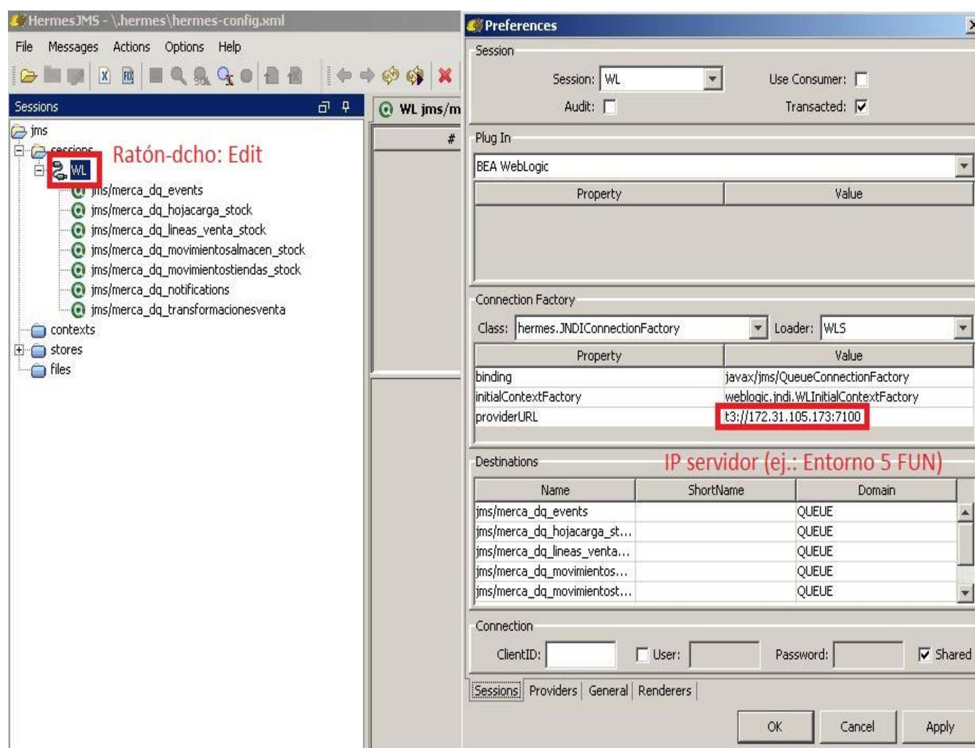


Figura 26. *Captura de la herramienta HermesJMS*

- En la *figura 26* se observa una captura donde se visualizan las colas de un servicio. En la parte izquierda están listadas todas las colas que contiene y en la parte de la derecha se muestra con un encuadre rojo dónde hay que indicar la dirección del servicio para que *HermesJMS* sea capaz de listar las colas.

  Para saber las colas a las que el servicio afecta, el tester ha de consultar el comportamiento del servicio que te indica el DTAN. En este punto, el tester ha de tener en cuenta que las colas almacenan durante unos segundos el mensaje que les llega del servicio hasta que *HermesJMS* los refresca automáticamente, por lo tanto, el tester deberá guardar además del log que genere el servicio, una captura de pantalla donde la cola tenga el mensaje (antes de que se refresque).

Por último, una vez que el tester haya ejecutado todos los casos funcionales y haya guardado tanto los logs como las capturas de pantalla del *HermesJMS*, si el servicio ha funcionado correctamente según el comportamiento indicado en el DTAN, el tester podrá dar como “OK” la fase funcional y pasar a la siguiente fase (la fase de seguridad explicada en la sección 4.5.1).

4.4.9 Script funcional (SCF) [STD privado]

Si el tester ha de ejecutar pruebas funcionales de un servicio standalone con operaciones privadas, va a tener que dejar de lado la herramienta *SoapUI* y tendrá que programar manualmente un script que haga uso del servicio (importándolo, esto se explica a continuación).

Una vez realizados el PLP y el ETF, será el momento de empezar a preparar el activo que permitirá al tester ejecutar las pruebas funcionales. Este activo tendrá la extensión "*.groovy*" [4] y es un activo bastante complejo así que la realización del fichero habrá que hacerla por secciones.

- En el DTAN se indicarán las clases que usa el servicio. Así que al principio del documento deberán importarse todos los servicios y clases que vaya a necesitar las operaciones del servicio (se muestra un ejemplo de la importación en la *figura 27*).

```
import es.mercadona.contratos.interfaz.ServicioCompleto_v1_0_0;
import es.mercadona.contratos.interfaz.ServicioCompleto_v0_0_1;
import es.mercadona.contratos.dto.serviciocompleto.*;
import es.mercadona.opvta.common.exception.*;
```

Figura 27. *Captura sobre las importaciones del SCF (STD privado)*

- Una vez importado las clases del servicio, el tester tendrá que crear un método por cada operación que tenga el servicio, pasándole como parámetro el identificador del caso (columna explicada en el apartado 4.4.3). Entonces, el tester ejecutará las pruebas llamando a estos métodos e indicando el caso que desea ejecutar.
  A continuación, se describen los pasos que debe seguir el método:

    - Lo primero que tiene que hacer el método es una comprobación de que el identificador del caso sea el correcto, sino lo es, el tester ha de indicar con un mensaje que hay un error con el identificador y se acabará la ejecución. En el caso de que sea correcto, el siguiente paso consistirá en cargar el ETF, de esta forma leerá los valores de cada parámetro según su identificador (tanto este paso como el siguiente están mostrados con un ejemplo en la *figura 28*).

    - El siguiente paso del método es indicar el servicio que se usará para la ejecución de las pruebas. Éste será uno de las importaciones que se ha hecho al principio del activo (véase la *figura 28*).

```
def operacionConsulta(numTC) {
    TIPO_ENTRADA = "entrada";
    TIPO_SALIDA = "salida";
    try{
        fichero = new FileWriter("./log/Log-AB12345_OP-01-FUN-TC_"+numTC+".log", true);
        pw = new PrintWriter(fichero);
        if(numTC >= 0){
            ETFentrada = load_file("./InOut/ETFentrada.csv");
            resultadoEsperado = load_file("./InOut/ETFsalida.csv");
            logEj("Parámetros de entrada del PLP son: " + ETFentrada);
            logEj("Parámetros de salida del PLP son: " + resultadoEsperado);
            service = ctx.getBean(servicioCompleto_v0_0_1);
            logEj("Se ejecuta el tc " + numTC);
```

Figura 28. *Método de la operación, SCF Standalone.*

- El método también tendrá que tener un "switch" (instrucción java que permite tomar uno de los diversos caminos de ejecución según el valor de una variable) con tantos casos como tenga la operación. En este punto, hay que distinguir dos tipos de casos:
    - Si el caso debe acabar con error, realizamos la llamada al método del servicio correspondiente dentro de un "try-catch" para capturar la excepción y, si esta ocurre, mostrar un mensaje para informar al Tester.

```
case 8:
    try{
        result = service.OperacionConsulta(datosTC);
        assert 1 == 2 : "No se ha lanzado la excepción ErrorBBDDException";
    }catch(ErrorBBDDException ebd){
        logEj("Excepcion -> ErrorBBDD");
        logEj("El mensaje amigable del error es: " + ebd.getMessage());
        assert ebd.getMessage() == "Error ejecutando una operación sobre base de datos."
        logEj("Los parametros son: " + ebd.getParams());
        logEj(ebd);
    }
    break;

case 9:
    try{
        result = service.OperacionConsulta(null);
        assert 1 == 2 : "No se ha lanzado la excepción ErrorFormatoParametrosEntrada";
    }catch(ErrorFormatoParametrosEntradaException efp){
        logEj("Excepcion -> ErrorFormatoParametrosEntrada");
        logEj("El mensaje amigable del error es: " + efp.getMessage());
        assert efp.getMessage() == "Parámetros de entrada con formato incorrecto. [Parámetros: OperacionConsultaIn]."
        logEj("Los parametros son: " + efp.getParams());
        logEj(efp);
    }
    break;

case 10:
    try{
        result = service.OperacionCompleto(null);
        assert 1 == 2 : "No se ha lanzado la excepción ErrorNoControlado";
    }catch(ErrorNoControladoException efp){
        logEj("Excepcion -> ErrorNoControlado");
        logEj("El mensaje amigable del error es: " + efp.getMessage());
        assert efp.getMessage() == "Parámetros de entrada con formato incorrecto. [Parámetros: OperacionConsultaIn]."
        logEj("Los parametros son: " + efp.getParams());
        logEj(efp);
    }
    break;

}
```

Figura 29. *Casos que se espera que acaben en error. SCF de Standalone*

- En la *figura 29* se controlan los tres errores que se muestra en el ejemplo del PLP y el ETF. Además de capturar la excepción (con la instrucción try-catch), se lanzan los mensajes pertinentes para que el tester sepa lo que ha ocurrido y tenga una mejor concepción del error ocurrido (tanto si es un error esperado, como si es debido a un fallo surgido durante el *testing*).

- Si el caso debe acabar en una salida sin errores, se deberá instanciar el método que usará la operación y guardar en una variable (en la *figura 29* se le ha denominado "result") lo que el servicio le ha devuelto al tester con los datos que le ha pasado (según la *figura 30,* estos datos se encuentran en la variable "datosTC" que con el método "declaración" los ha cogido del ETF).

```
def declaracion(numTC){

    /** Instanciar la entrada **/

    input = new operacionConsulta();

            input.setLocaleLanguage(ETFen[numTC][1].get("localeLanguage"));
            input.setLocaleCountry(ETFen[numTC][1].get("localeCountry"));
            input.setLocaleVariant(ETFen[numTC][1].get("localeVariant"));

            input.setCodProveedor(ETFen[numTC][1].get("codProveedor"));
            input.setCodProducto(ETFen[numTC][1].get("codProducto"));
            input.setFechaEntrega(ETFen[numTC][1].get("fechaEntrega"));
            input.setFechaCaducidad(ETFen[numTC][1].get("fechaCaducidad"));

    logEj("Entra en declaracion " + numTC);
    logParam(TIPO_ENTRADA, input);
    input;

}
```

Figura 30. *Ejemplo del método declaración. SCFStandalone*

- Finalmente, cada caso deberá escribir un archivo donde guardará el log, uno de los ficheros más importantes del tester. El nombre del fichero deberá tener la operación y el caso para que más tarde el tester pueda diferenciar qué fichero corresponde a qué caso.

```
if(numTC > 0){
    datosTC = declaracion(numTC);
    switch(numTC){
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
    case 6:
    case 7:
        result = service.OperacionConsulta(datosTC);
        comprobarResultados(result, numTC);
        break;
```

Figura 31. *caso sin errores. SCFStandalone*

- En la *figura 31* se observa un ejemplo de SCF que continúa con el ejemplo del PLP. Los siete casos lanzarán el servicio (“datosTC” tendrá los valores del caso que indique el PLP) y guardarán la respuesta en un fichero al que se ha denominado “result” y, como se explica a continuación, se le ha agregado una llamada al método “comprobarResultados”.

- El último método que el tester puede agregar (es opcional pero útil para automatizar en mayor medida el proceso del *testing*), es un método que compruebe los resultados. Ésto se consigue guardando la salida del método que se llama en cada caso en una variable. Finalmente, cargando el ETF de salida (explicado en el apartado 4.4.3) en otra variable, se puede comprobar una salida con la otra devolviendo un resultado booleano (“true” o “false”). El tester habrá de revisar los logs igualmente para asegurarse, pero con este método puede revisarlos al final de todas las ejecuciones si éstas son correctas (es decir, si el método devuelve “true”).

```
def comprobarResultados(resultadoReal, numTC){
    resultadoEsperado=resultadoEsperado[numTC];
    logEj("Entra en comprobarResultados Test case: " + numTC);

    assert resultadoReal.getLocaleLanguage() == resultadoEsperado[1].get("localeLanguage");
    assert resultadoReal.getLocaleCountry() == resultadoEsperado[1].get("localeCountry");
    assert resultadoReal.getLocaleVariant() == resultadoEsperado[1].get("localeVariant");
    assert resultadoReal.getCodProveedor() == resultadoEsperado[1].get("codProveedor");
    assert resultadoReal.getCodProducto() == resultadoEsperado[1].get("codProducto");
    assert resultadoReal.getFechaEntrega() == resultadoEsperado[1].get("fechaEntrega");
    assert resultadoReal.getFechaCaducidad() == resultadoEsperado[1].get("fechaCaducidad");
}
```

Figura 32. *método para comprobar resultados. SCF Standalone*

- La *figura 32* es un ejemplo del método y, básicamente se hace una comprobación por cada parámetro teniendo en cuenta el caso que se ha ejecutado y el resultado esperado que se ha extraído del activo PLP.

Como se puede observar, la realización de este documento es costoso (aunque se pueden hacer plantillas para agilizar el proceso), pero tiene la gran ventaja de que el tester puede introducir mensajes por pantalla que le especifiquen qué se está ejecutando y cual puede haber sido el fallo (la personalización de documentos, aunque a corto plazo puede llevar tiempo, a largo plazo es rentable y el *testing* mejora).

4.4.10 Ejecución funcional (BATCH)

Si el tester ha de ejecutar un servicio de tecnología BATCH, tendrá que usar un nuevo programa que le permita hacer las ejecuciones del servicio y poder realizar las pruebas funcionales.

Para la ejecución de estas pruebas, el tester usa le herramienta *Putty* (explicada en el apartado 3.3). Es una de las herramientas más utilizadas en servicios de framework BATCH debido a su facilidad de uso. Cuando el servicio es de framework BATCH no necesita ningún tipo de ETF que ayude a la herramienta para su ejecución (esto se debe a que los servicios que usan tecnologías BATCH no suelen tener parámetros de entrada y, de tenerlos, solo tiene uno con valores muy concretos), el tester solo ha de conectarse mediante SSH al entorno del cliente del servicio y lanzarlo. Una vez que se esté ejecutando el servicio sin ningún tipo de error, el tester tendrá que abrir otra conexión *Putty*, pero esta vez de tipo *Telnet [3]*, y será sobre ésta ventana donde ejecutaremos el servicio, cada vez con un valor de entrada distinto en el caso de que lo tenga.

En este caso, los logs son capturas de pantalla y copias del código de la ventana del SSH donde veremos su salida. En caso que se realice acciones sobre ficheros también se visualizará en la ventana (esto puede ocurrir según la funcionalidad del servicio, explicado en la sección 4.4.5).

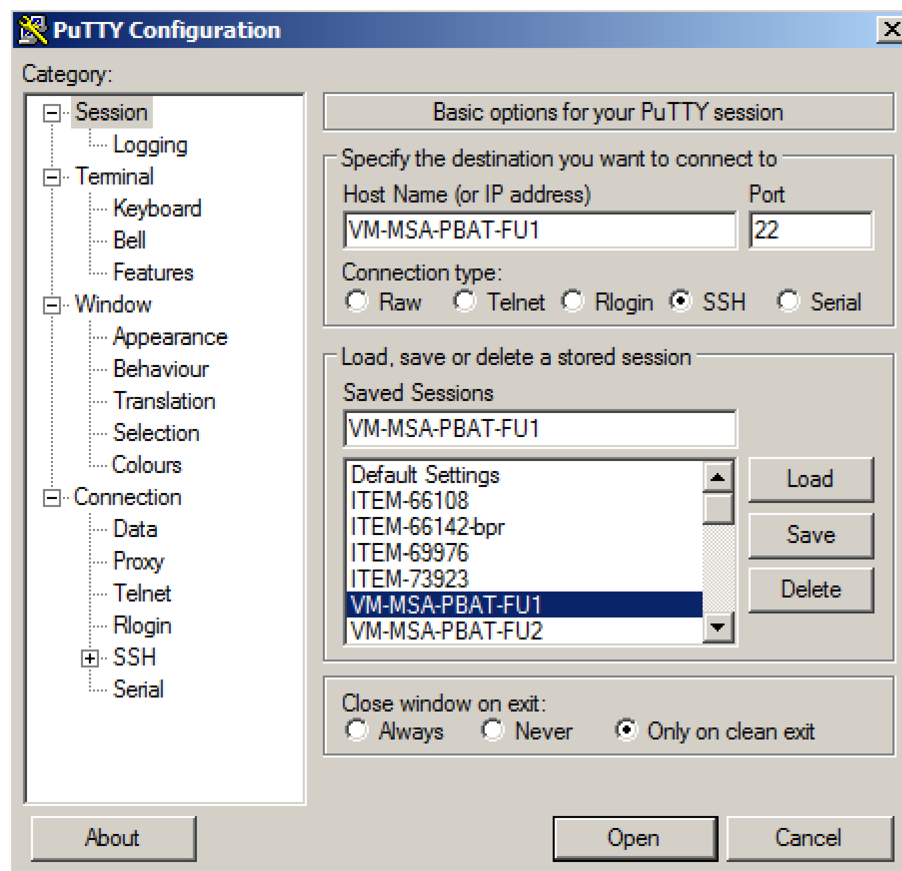


Figura 33. *captura de la herramienta Putty*

- En la *figura 33* se visualiza la herramienta Putty. En este caso, por motivos de seguridad no se puede mostrar un ejemplo de ejecución dado que solo serían terminales con datos privados sobre un servicio de una empresa real.
- Lo que se puede observar es que la herramienta permite cambiar la conexión de SSH a Telnet con un simple click. Además de permitirnos guardar una lista de conexiones por si en posibles futuros cambios de fase de la petición, hay que volver a conectarse.

- Por último, hay que señalar que dada la falta de activos y la poca participación del tester en preparar las pruebas, si algo funciona mal durante la ejecución, es probable que tan solo pueda reportar el fallo y no sea capaz ni de solucionarlo ni de especificar el problema (aunque si el servicio hace uso del DAT, el tester tiene algo con lo que trabajar y así, proporcionarle algo de ayuda al cliente si no ha ido correctamente).

  Estos servicios, aunque podría realizar el *testing* la propia empresa, es común que prefieran que lo realice una empresa especialidad en el *testing* y tener un "sello" de garantía sobre la calidad del servicio.

4.5 Fase de pruebas de seguridad

Una vez acabada con éxito la fase funcional, el tester deberá asegurarse de que, además de que los parámetros se comportan de forma correcta, también tiene un nivel de seguridad adecuado.

Para testear el nivel de seguridad, el tester necesitará un proyecto SoapUI (activo SCS ya explicado en el apartado 4) que realice unas pruebas de seguridad según el tipo de los parámetros y adaptándose a cada uno de ellos.

Antes de continuar, hay que mencionar las razones por las que, ni en los servicios BATCH ni en los servicios con framework standalone, se realizan estas pruebas.

- La principal razón es debido a que no tienen parámetros de entrada o, en el caso de tenerlos, siempre son pocos y con valores fijos (por ejemplo, que solo admita como usuario, "admin" o "user").

- La otra razón principal es: debido a que son servicios locales y, cualquier fallo de seguridad que no se origine por los parámetros de entrada, será por algún fallo que se produzca en el sistema local donde se ha implantado el servicio (algo totalmente externo al servicio a testear y, puede que también externo al cliente).

4.5.1 Script de seguridad (SCS) [FW online, STD público]

Una vez que el tester haya dado "OK" funcional, puede empezar con las pruebas de seguridad, pruebas que realizará con el mismo programa que con el que ha ejecutado las pruebas funcionales (*SoapUI*) y que le permitirán saber si el servicio tiene el suficiente nivel de seguridad como para no generar problemas ni a los usuarios ni al propietario del servicio.

El siguiente paso del tester en los servicios online es la fase de seguridad. En este caso, la herramienta *SoapUI* tiene un "step" especial para realizar este tipo de pruebas y que lo hace extremadamente útil. Este "step" es de seguridad y el tester ha de crear dos subtipos

del “step”: uno de ellos controlará los parámetros de tipo string y fecha, el otro controlará todos los parámetros (Tanto parámetros de caracteres como numéricos).

En este caso, el *SoapUI* automáticamente guardará el log de seguridad y generará un informe. Este informe verificará lo que se ha comprobado en las pruebas funcionales con una batería de pruebas más amplias para cada uno de los parámetros. Por lo general, si las pruebas funcionales han terminado con éxito, este proyecto solo confirmará al tester de que, funcionalmente, puede ser lanzado al mercado (aunque hasta que no se pasen las pruebas de rendimiento no se puede confirmar su fluidez ni su nivel calidad).

En el caso de producirse un error, significará que hay un caso funcional que ha fallado o se le ha pasado al tester. Por lo tanto, tendrá que volver a la fase de pruebas funcionales y encontrar el error (en estos casos, resulta útil al tester revisar los logs para encontrar el fallo y, corregirlo o notificarlo), y si el error no afecta a todos los casos, el tester no tiene por qué realizar de nuevo la fase completa, aunque si deberá repetir los casos que puedan haberse visto afectado por el error (un buen tester debe comprender el error para distinguir aquellos casos afectados directa o indirectamente por el fallo de los que no).

- Ejemplo Real (En los apartados que se necesite el activo WSDL se ha usado imágenes de servicios reales para la explicación)

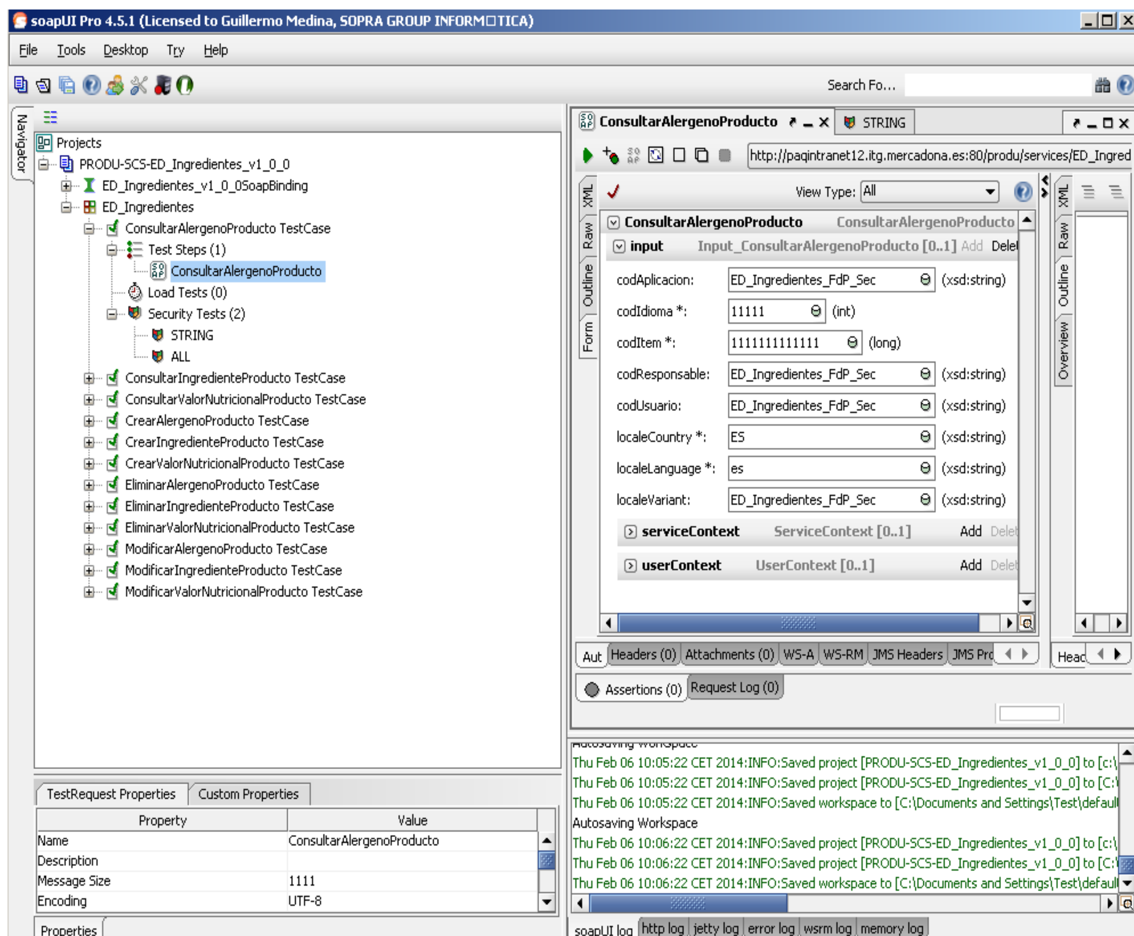


Figura 34. *ejemplo de SCS con sus correspondientes datos.*

- En la *figura 34* se observa el "test Request" y los dos "test" de seguridad. En el "test String" se introducen valores aleatorios forzando el máximo su rango:
  - Por ejemplo, el parámetro "codIdioma" al ser un entero de como máximo 5 dígitos, se introduce 5 dígitos aleatorios (que durante la ejecución variará su valor entre 0 y 5 dígitos).
  - El valor en las pruebas de seguridad será irrelevante, porque una vez indicado el rango máximo, la herramienta *SoapUI* empezará a realizar pruebas aleatorias con valores aleatorios para verificar la seguridad de cada campo/parámetro.
- Por último, como se ha mencionado anteriormente, dada la complejidad que puede suponer el formato fecha y la variedad de caracteres que se pueden introducir a los parámetros de tipo "string", estos tienen un test extra. Además de incluirse junto con el resto de campos en el test de seguridad "ALL"

4.6 Fase de Rendimiento

Ahora que se ha verificado que el servicio funciona de forma correcta y tiene un buen nivel de seguridad, hay que comprobar el rendimiento del servicio. Esta comprobación se realiza mediante: un activo para llenar las tablas de datos con grandes cantidades de registros (DAR), un activo con una ingente cantidad de valores de entrada (ETR) y un proyecto(SCR) que recopile los anteriores activos mencionados y permita al tester medir los tiempos del servicio (estos activos han sido explicados brevemente en el apartado 4).

Antes de comenzar con esta fase, hay que destacar que BATCH no requiere de ningún activo y, que desde la terminal (con comandos como "time", indica al usuario el tiempo que ha tardado la llamada) se puede realizar un rendimiento. Esto lo puede hacer el cliente o éste le debe dar acceso al tester para poder ejecutar de forma remota el servicio en el entorno donde se desplegará (al igual que en standalone, hay que ser conscientes de que la mejor forma de probar el rendimiento de los servicios locales es en el entorno donde se desplegará). El tester deberá guardar capturas de la ejecución del terminal indicando el tiempo de la duración de cada llamada (además puede realizar un promedio o un percentil 90 de los tiempos si el cliente lo desea, esto tiene que estar definido en el DTAN o se tiene que haber decidido en el apartado 4.1, preproceso del *testing*).

En esta fase el tester ha de saber si los tiempos del servicio son adecuados y aceptables para el cliente (estos tiempos han de estar acordados en el preproceso del testing y lo mejor es que el cliente haya realizado un estudio de mercado para saber qué tiempos de respuesta tienen servicios similares o, si no hay, que tengan una carga similar).

Existen cuatro tipos de pruebas de rendimiento que hay que ejecutar para asegurar que el servicio responderá adecuadamente en todas las situaciones en las que se pueda encontrar:

- **Prueba de carga** (duración 5 minutos): En este tipo de pruebas, el servicio se ejecutará con los hilos concurrentes que tendrá el servicio de media, es decir, se ejecutará el servicio con una carga similar a la real (se muestra una gráfica de la prueba en la *figura 35*).

En este caso, el tester tendrá que consultar el cuadro de rendimiento del DTAN y ver los usuarios medios que estima el cliente que tendrá el servicio.

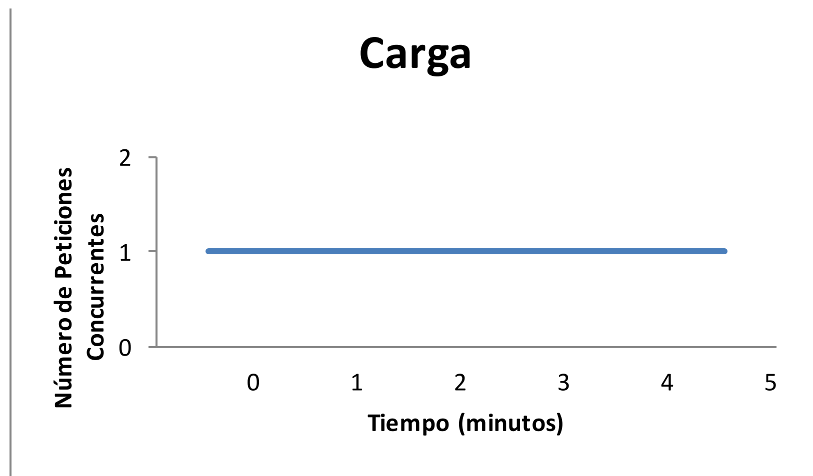


Figura 35. *grafica de peticiones en una prueba de rendimiento (carga)*

- **Prueba de estrés** (duración 1 minuto): En esta prueba, el servicio se ejecuta con el número máximos de hilos que contempla el servicio (se muestra una gráfica de la prueba en la *figura 36*). De esta forma, el tester se asegurará que los tiempos son aceptables incluyendo en el umbral de los recursos. (es decir, cuando el servicio usa todos los recursos que tiene disponibles y maneja el máximo de peticiones que es capaz de controlar, según el cuadro de rendimiento del DTAN, véase el apartado 4.2)
  Es frecuente que los servicios tengan picos anormales de actividad (cuando hay ofertas en una página web o más trabajadores de lo normal acceden al mismo tiempo al servicio debido algún suceso puntual), en estas situaciones se confirma si el servicio tiene la calidad suficiente como para soportar esta actividad (esta es una de las pruebas más críticas del *testing* y donde el servicio suele fallar por recibir demasiadas peticiones o estar mucho tiempo funcionando a máxima potencia).

Figura 36. *grafica de peticiones en una prueba de rendimiento (estrés)*

- **Prueba de escalabilidad** (duración 10 minutos): Esta prueba sirve fundamentalmente para comprobar que el servicio se adapta a los hilos que va recibiendo, es decir, empieza la ejecución del servicio con los hilos que tendrá de media y lo aumentas hasta que tenga el máximo de hilos que contempla el servicio (indicado en el DTAN, la *figura 37* muestra una gráfica de la prueba en cuestión). Aunque el servicio haya sido capaz de gestionar el máximo número de hilos que se estima que tendrá el servicio (prueba de estrés anteriormente explicada), no significa que sea capaz de flexibilizar sus recursos según los cambios del tráfico del servicio, así que el tester tendrá que realizar este test independientemente del resultado de la prueba de estrés.

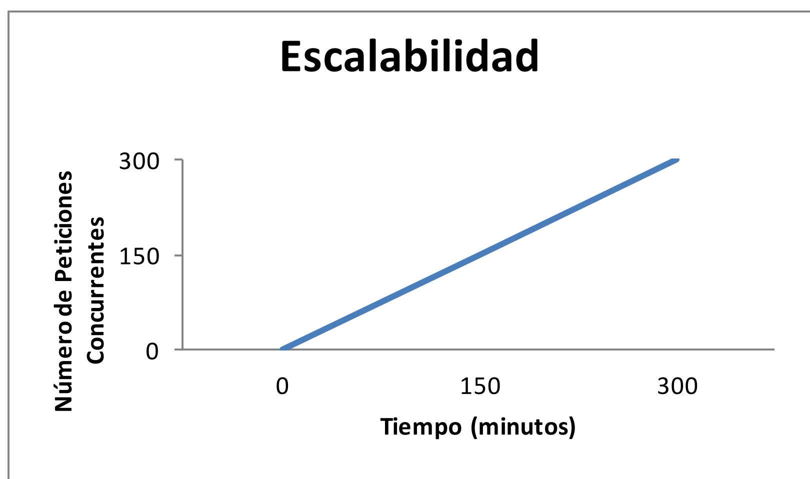


***Figura 37.** grafica de peticiones en una prueba de rendimiento (escalabilidad)*

- **Prueba de estabilidad** (30 minutos): Esta es una prueba de carga de larga duración para corroborar que el servicio es estable durante un periodo relativamente largo (se muestra una gráfica de la prueba en la *figura 38*).
  Por norma general, esta prueba no suele dar fallos, pero es recomendable hacerlo para que el tester se asegure que el servicio es capaz de proporcionar unos recursos constantes y soporte un tráfico medio de forma continuada y sin errores.

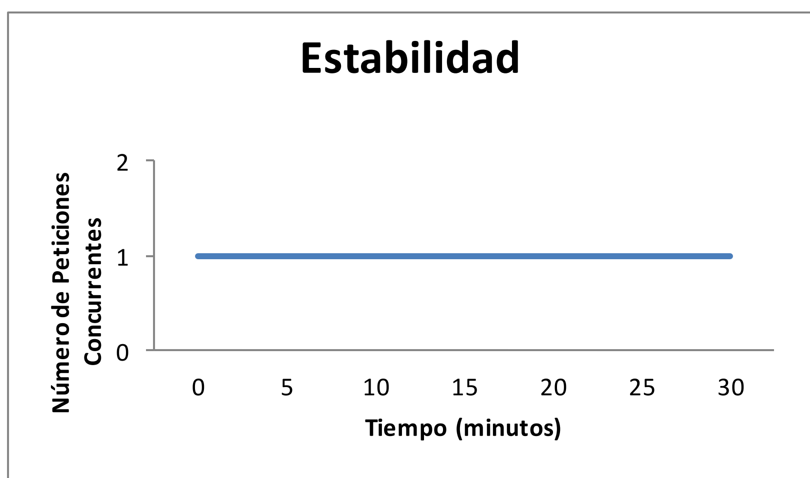


***Figura 38.** grafica de peticiones en una prueba de rendimiento (estabilidad)*

4.6.1 Documento de alto rendimiento (DAR) [FW2,FW3,STD público]

Una vez descritas todas las pruebas de rendimiento, el tester tendrá que empezar a preparar los activos que necesite para la ejecución, una ejecución que, de una forma u otra, le llevará al fin del *testing* (para bien o para mal).

El primer activo que el tester ha de preparar es el DAR, un activo que dependiendo de lo que haga el servicio, puede no ser necesario (es un activo que solo es necesario si el servicio interactúa con la base de datos y, al igual que el DAT, puede complicarse en gran medida para el tester).

En el caso de que el servicio interactúe con la base de datos, este activo se encargará de llenar las tablas que use el servicio.

Este fichero tendrá todas las tablas necesarias con las columnas que son clave primaria (columnas obligatorias de una tabla) y las que usa el servicio (aquellas columnas que no sean necesarias ni sean claves primarias, no se ponen ya que complicarían el activo innecesariamente).

Para cada una de las columnas de las tablas tendrá que indicarse: un valor inicial, un incremento o decremento de ese valor y el número de registros que se insertarán en la tabla.

De esta forma el tester conseguirá con el activo llenar las tablas con cientos o miles de registros de forma automática. La razón por la que se realiza ese incremento o decremento es: para poder distinguir un registro de otro, porque las claves primarias no pueden repetirse y, sobre todo, para que la caché no afecte a la prueba (si dos registros son completamente iguales, el servicio usará la caché invalidando los tiempos de respuesta y, por lo tanto, la ejecución).

Este es uno de los activos más complicados del proceso al testear una aplicación debido a las complicaciones que puede haber entre las tablas y su relación entre ellas. Pero a pesar de lo complicado que puede volverse, esto es necesario para simular una situación real en la base de datos, donde el servicio ha de buscar lo más rápido posible aquellos registros que cumplen ciertas condiciones en una tabla llena.

4.6.1.1 Ejemplo explicativo

Siguiendo con el ejemplo del PLP y ETF, la *figura 39* sería un buen ejemplo del activo DAR. En la *figura 39* aparecen las tres tablas necesarias con sus columnas, para cada columna se indica: su nombre, tipo y un valor inicial. Además, si el valor se va ir incrementando (por razones anteriormente explicadas) necesitará: una frecuencia, un valor mínimo y un valor máximo.

Al final de cada tabla se indica los registros que tendrá la tabla en cuestión (esto deberá indicarse en la base de datos que aparecerá en el DTAN).

```xml
<name>Aplicacion</name>
<table>
    <name>TABLA_PROVEEDORES</name>
    <columns>
        <column>
            <name>COD_N_PROVEEDOR</name>
            <type>Varchar2</type>
            <init>Proveedor 1</init>
            <incremental>true</incremental>
            <frequency>1</frequency>
            <minValue>false</minValue>
            <maxValue>false</maxValue>
        </column>
        <column>
            <name>TXT_PROVEEDOR</name>
            <type>Varchar2</type>
            <init>proveedor de bebidas</init>
            <incremental>false</incremental>
        </column>
    </columns>
    <nrows>10000</nrows>
</table>

<table>
    <name>TABLA_PRODUCTO</name>
    <columns>
        <column>
            <name>COD_N_PRODUCTO</name>
            <type>Number</type>
            <init>1000000002</init>
            <incremental>true</incremental>
            <frequency>1</frequency>
            <minValue>false</minValue>
            <maxValue>false</maxValue>
        </column>
        <column>
            <name>TXT_PRODUCTO</name>
            <type>Varchar2</type>
            <init>Agua</init>
            <incremental>false</incremental>
        </column>
    </columns>
    <nrows>10000</nrows>
</table>
```

```xml
<table>
    <name>TABLA_FINAL</name>
    <columns>
        <column>
            <name>COD_N_PROVEEDOR</name>
            <type>Varchar2</type>
            <init>Proveedor 1</init>
            <incremental>true</incremental>
            <frequency>2</frequency>
            <minValue>false</minValue>
            <maxValue>false</maxValue>
        </column>
        <column>
            <name>COD_N_PRODUCTO</name>
            <type>Number</type>
            <init>1000000002</init>
            <incremental>true</incremental>
            <frequency>1</frequency>
            <minValue>1000000002</minValue>
            <maxValue>1000010002</maxValue>
        </column>
        <column>
            <name>FECHA_ENTREGA</name>
            <ctrlValue>TRUNC(SYSDATE)</ctrlValue>
            <incremental>false</incremental>
        </column>
        <column>
            <name>FECHA_CADUCIDAD</name>
            <ctrlValue>TRUNC(SYSDATE+1)</ctrlValue>
            <incremental>false</incremental>
        </column>
    </columns>
    <nrows>15000</nrows>
</table>
```

Figura 39. *ejemplo del activo DAR*

Para realizar este fichero el tester puede encontrarse con uno de los problemas más complicados que puede surgir: la tabla proveedores y producto tienen una volumetría de 10.000 registros y, sin embargo, la tabla final tiene 15.000 registros. Debido a que el tester ha de llenar la tabla completa pero no tiene 15.000 registros distintos ni de producto ni de proveedores, hay que combinarlos.

La forma de hacer esto es con las claves primarias. La tabla final tiene dos claves primarias, así que la solución es establecer una frecuencia de 2 para que así usemos 7500 registros de proveedor, y de la clave principal de producto se usarán los 10.000 registros (luego volverá a su valor inicial y seguirá incrementándose). Esta es una forma de solucionarlo, pero se puede complicar si intervienen más tablas y claves primarias.

Al igual que el DAT, si las tablas tienen claves ajenas y, estas tablas tienes otras claves ajenas, se puede complicar de forma exponencial el activo. El tester tendrá que ser muy cuidadoso durante la realización del activo o tendrá que volver a empezar la fase de rendimiento desde el principio.

4.6.2 Entrada de rendimiento (ETR) [FW2,FW3,STD público]

Si el tester ha logrado hacer el activo DAT (de haber sido necesario), tendrá que empezar a preparar el ETR, un activo que se encargará de asociar (de una forma ligeramente distinta al ETF) valores a cada parámetro.

Este activo es un fichero Excel (se hace de esta manera debido a que buscamos asociar cientos o miles de valores a cada parámetro, hacerlo como el ETF llevaría demasiado tiempo) con la extensión ".CSV" donde tendrá una entrada con los valores de los parámetros (cuyo valor irá variando para distinguir unos registros de otros) y habrá tantos registros como vaya a necesitar el servicio (para todas y cada una de las pruebas explicadas en la sección 4.6).

El número de registros se puede calcular como: el tiempo que tarda en realizar una petición por el número de peticiones que recibirá el servicio (indicado en el cuadro de rendimiento del DTAN y que varía según la prueba de rendimiento que se use) por el tiempo que dure la ejecución (tiempo que varía según la prueba de rendimiento)

4.6.2.1 Ejemplo Explicativo

Siguiendo con el ejemplo del PLP, la *figura 40* muestra un ejemplo del ETR (en los dos formatos anteriormente mencionados).

Este activo se ha abierto de dos maneras: en la parte derecha de la *figura 40* se ha abierto mediante la herramienta microsoft excel y en la parte izquierda se ha abierto con un editor de textos (esto es necesario para eliminar la última línea que te genera de forma automática el excel). Se han usado dos columnas, cada una corresponde a los dos parámetros cuyo valor varía (aunque el valor del primer parámetro varíe cada dos registros, con el segundo parámetro se puede distinguir el primer registro del segundo).

```
1   Proveedor 1;1000000002
2   Proveedor 1;1000000003
3   Proveedor 2;1000000004
4   Proveedor 2;1000000005
5   Proveedor 3;1000000006
6   Proveedor 3;1000000007
7   Proveedor 4;1000000008
8   Proveedor 4;1000000009
9   Proveedor 5;1000000010
10  Proveedor 5;1000000011
11  Proveedor 6;1000000012
12  Proveedor 6;1000000013
13  Proveedor 7;1000000014
14  Proveedor 7;1000000015
15  Proveedor 8;1000000016
16  Proveedor 8;1000000017
17  Proveedor 9;1000000018
18  Proveedor 9;1000000019
19  Proveedor 10;1000000020
20  Proveedor 10;1000000021
21
```

	A	B
1	Proveedor 1	1000000002
2	Proveedor 1	1000000003
3	Proveedor 2	1000000004
4	Proveedor 2	1000000005
5	Proveedor 3	1000000006
6	Proveedor 3	1000000007
7	Proveedor 4	1000000008
8	Proveedor 4	1000000009
9	Proveedor 5	1000000010
10	Proveedor 5	1000000011
11	Proveedor 6	1000000012
12	Proveedor 6	1000000013
13	Proveedor 7	1000000014
14	Proveedor 7	1000000015
15	Proveedor 8	1000000016
16	Proveedor 8	1000000017
17	Proveedor 9	1000000018
18	Proveedor 9	1000000019
19	Proveedor 10	1000000020
20	Proveedor 10	1000000021

Figura 40. *ejemplo del activo ETR*

En la *figura 40* se ha simplificado a 20 registros, pero el número de registros necesarios dependerá de: los hilos que use el servicio, el número máximo de registros que es capaz de devolver la consulta y el tiempo entre las consultas ("delay", este tiempo se controlará en el SoapUI y quedará reflejado en el informe de rendimiento, se muestra en el apartado 4.6.3).

Para finalizar, el activo se hace mediante el excel debido a que nos permite escribir cientos de valores en poco tiempo gracias a la secuencialidad que tiene integrada el excel. El problema es que al guardarlo crea una última línea vacía que afecta al rendimiento, esto se soluciona como se ha mencionado anteriormente (eliminando la última línea con un editor de textos).

4.6.3 Script de rendimiento (SCR)[FW2,FW3,STD público]

Cuando el tester ya dispone de los dos activos anteriormente citados, el tester podrá empezar a preparar el SCR.

El proyecto *SoapUI* se creará igual que en la fase funcional (apartado 4.4.7): el tester creará el proyecto asignándole el activo WSDL e introducirá para cada operación (testSuit) un caso con los parámetros.

A continuación, al proyecto de rendimiento se le agrega el paso "ETR", de esta forma se podrá enlazar cada parámetro con cada valor del ETR.

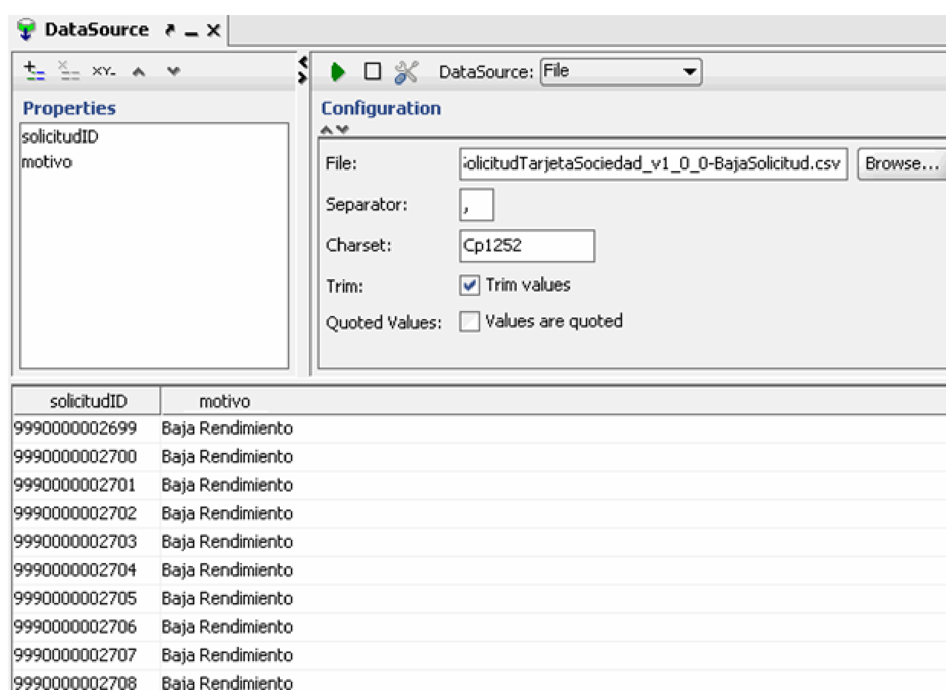


Figura 41. *Captura del ETR en el activo SCR*

En la *figura 41* se observa en la parte derecha cómo se ha seleccionado el archivo .CSV y en la parte inferior se han cargado la lista de valores para los dos parámetros que tiene el servicio y el activo (dos parámetros que se han definido como propiedades en la parte izquierda).

Finalmente, se le agregará al SCR el paso "delay" para que controle el tiempo entre las peticiones (tiempo que cambia según la prueba de rendimiento que se esté realizando).

Figura 42. *Conjunto de test de un SCR*

- En la *figura 42* se visualiza: el test "Datasource" que habrá cargado los datos del ETR, el test "ConsultarSolicitud" que es el test Request (tendrá únicamente los parámetros del Datasource enlazados y el resto se establecerá un valor fijo), el Delay que permitirá controlar los tiempos entre peticiones y, finalmente, tiene el test "DataSource Loop" que hará un bucle cogiendo cada vez una línea distinta del Datasource/ETR.

Una vez realizados los "test" y el tester haya ejecutado las pruebas de rendimiento, se informa al cliente del servicio o se consulta el estándar, si lo tiene, para ver si los tiempos son adecuados. Si son correctos se puede pasar a la entrega, y en el caso de que no lo sean, el tester se lo notificará al cliente y buscará la forma más plausible de hacerlo viable. Por lo general, se soluciona este problema reduciendo el tamaño de la tabla, el número de los registros o las peticiones que el servicio puede soportar para conseguir un tiempo aceptable (en estos casos solo habría que repetir la fase de rendimiento, ya que no afectaría en ningún momento ni a la funcionalidad ni a la seguridad del servicio).

4.6.3.1 Ejemplo Explicativo

En *la figura 43* se visualiza la salida de un SCR. Una vez obtenido se consultará qué tiempos son aceptables según el cliente y el estándar del servicio. Si son correctos se habrá terminado y si no, habrá que hacer modificaciones que mejoren los tiempos (modificaciones ya mencionadas anteriormente).

carga

Threads: 2 Strategy Simple Test Delay 0 Random 0.5

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
DataSource	23	23	23	23	2	8,26	0	0	0	0
ConsultarSolicitud	98	9223	306,38	125	200	3,33	429564	7170	0	0
Delay [300]	277	315	292,65	284	198	3,31	0	0	0	0
DataSource Loop	1	1	0	0	198	3,31	0	0	0	0
TestCase:	37721	963146	41.048	41048	1	0,01	43385964	724173	0	0

Figura 43. *test de carga de un SCR*

Para finalizar la explicación y dejarla más clara, en la *figura 44* se observa la fase de rendimiento simplificada y la implicación de cada activo en la fase.

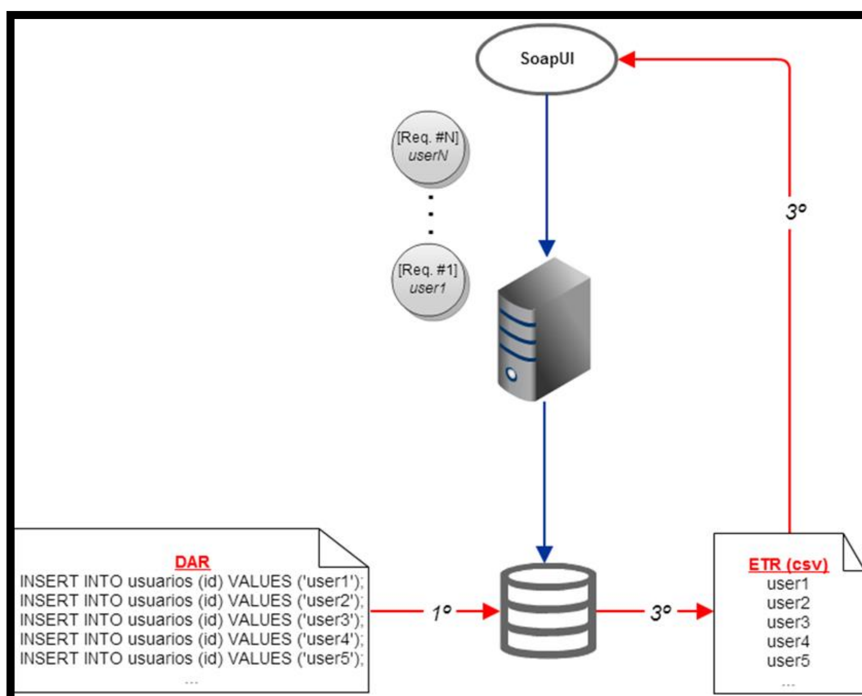


Figura 44. *simplificación de la fase de rendimiento*

4.6.4 Script de rendimiento (SCR) [STD privado]

En el caso de que el tester tenga que realizar el rendimiento de un servicio con framework standalone, el tester tendrá que copiar el SCF (este activo ha sido definido y explicado en la sección 4.4.9) y modificarlo ligeramente para obtener resultados propios de la fase de rendimiento.

Uno de los cambios se realizará en la operación principal de consulta, en esta operación en lugar de cargar el ETF de entrada, el tester deberá cargar el activo ETR (dichos cambios están marcados en la *figura 45*). Además, el tester no necesitará el resultado esperado, esto se debe a que, en la fase de rendimiento, a diferencia de la fase funcional, el tester se centra en los tiempos del servicio y no en los valores que éste devuelve.

```
def operacionConsulta(numTC) {
    TIPO_ENTRADA = "entrada";
    TIPO_SALIDA = "salida";
    try{
        fichero = new FileWriter("./log/Log-AB12345_OP-01-FUN-TC_"+numTC+".log", true);
        pw = new PrintWriter(fichero);
        if(numTC >= 0){
            ETFentrada = load_file("./InOut/ETFentrada.csv");
            resultadoEsperado = load_file("./InOut/ETFsalida.csv");
            logEj("Parámetros de entrada del PLP son: " + ETFentrada);
            logEj("Parámetros de salida del PLP son: " + resultadoEsperado);
            service = ctx.getBean(servicioCompleto_v0_0_1);
            logEj("Se ejecuta el tc " + numTC);
```

Figura 45. *SCF de STD con marcas que hay que cambiar para el SCR*

El siguiente cambio que el tester ha de hacer, afecta directamente a la ejecucion (véase en la *figura 46*). El tester ha de crear un bucle con tantas llamadas como datos tiene el ETR,marcar un delay (en el apartado 4.6 se ha explicado el delay) y, lo mas importante, el tester deberá almacenar los dos tiempos mas relevantes de la ejecucion:

- la duracion de cada llamada que se haga (estas se almacenarán en una lista).
- La duración del conjunto de llamadas incluyendo el tiempo del delay.

```
llamadas = ETR.size();
delay = 300;
start = System.currentTimeMillis();
def time_aux_ex = new Long[ETR.size()-1];
while(llamadas > 0)
{
    llamadas = llamadas-1;
    antes = System.currentTimeMillis();
    result = service.OperacionConsulta(datosTC);
    despues = System.currentTimeMillis();
    time_aux_ex.putAt(despues-antes);
    sleep(delay);
}
final = (System.currentTimeMillis() - start)/1000;
```

Figura 46. *llamadas al servicio de un SCR con fw Standalone privado*

Por último, el tester deberá almacenar los tiempos de rendimiento (véase en la *figura 47*) dejando constancia del delay utilizado y la fecha en la que se ha realizado el *testing* de rendimiento (como se ha mencionado durante el trabajo, estos ficheros que se almacenan con los datos de las pruebas del *testing* es lo más importante para el tester al dejar constancia de lo que se ha hecho y cómo ha respondido el servicio a cada una de las pruebas).

```
//Generar fichero fin ejecucion de Rendimiento
    pr.println("\n\n");
    pr.println("RESUMEN DE LA PRUEBA REALIZADA " + nombre_peticion);
    pr.println("Inicio de la prueba: " + diaActual_ini + " " + horaActual_ini);
    pr.println("Fin de la prueba: " + diaActual_fin + " " + horaActual_fin);
    pr.println("Tiempo de ejecucion: " + (final-start));

//Calculo del tiempo medio por peticion
            pr.println("Numero de peticiones: " + ETR.size());
            pr.println("Delay: " + delay);
            pr.println("\t\tiempo medio: " + (final-start)/ETR.size() + " ms");

//Calculo del percentil90
        int[] time_ex_sort = Arrays.copyOfRange(time_ex, 0, aux_i);;
        Arrays.sort(time_ex_sort);
        int aux = Math.rint(0.9 * time_ex.size());
        int P90 = time_ex_sort[aux-1];
        pr.println("\t\tPercentil90: " + P90 + " ms");
        pr.println("\t\tMAX: " + time_ex_sort[time_ex_sort.size()-1] + " ms");
        pr.println("\t\tMIN: " + time_ex_sort[0] + " ms");
        file.close();
```

Figura 47. *Guardar los logs del activo SCR de un servicio Standalone privado*

El tester debe ser consciente que los datos los va a leer el cliente y, por lo tanto, es necesario que muestre los tiempos como una media de cada llamada y un percentil 90(si el cliente desea almacenar otros tiempos, debería decirlo en el preproceso del *testing* ya que no es lo habitual).

4.7 Entrega y reflexiones del testing

Después de todo el proceso del *testing*, todos los activos de cada fase y las pruebas que se han realizado sobre el servicio, al tester sólo le queda entregar los resultados del proceso al cliente.

En la entrega, el tester ha de asegurarse de entregarle todos los documentos que ha ido creando durante el proceso y todos los logs que se han generado durante las ejecuciones. Destacando:

- **Fase funcional:** el activo PLP con los resultados de cada caso y las observaciones del tester (una sección muy importante indicado en el apartado 4.4.1).

- **Fase de seguridad:** el informe que genera el *SoapUI* por cada "test" de seguridad (aunque los logs son importantes, el informe generado es más fácil de interpretar, y si el cliente quiere datos más precisos, tendrá los otros logs).

- **Fase de rendimiento:** capturas de la herramienta *SoapUI* con los resultados de cada prueba de rendimiento (indicado en el apartado 4.6, aunque hay que entregarle los logs que el *SoapUI* devuelve por si el cliente quiere datos más concretos).

Durante el trabajo se han visto: las fases en las que se divide el *testing*, los diversos activos que tiene cada una de ellas con sus peculiaridades/complejidades (que pueden variar según el framework del servicio) y los resultados/logs que genera cada ejecución.

Todos estos pasos demuestran el enorme trabajo que lleva testear un servicio y lo complicado y arduo que puede resultar hacerlo, sin contar la cantidad de detalles que hay en cada fase que obliga a que el tester sea un experto en cada una de las herramientas y activos a realizar sino quiere realizar un *testing* incompleto (debido a validaciones o funcionalidades del servicio que el tester no haya testeado o parámetros que no haya forzado su rango para comprobar su nivel de seguridad).

La última reflexión a tener en cuenta sobre el proceso del *testing* es, que un fallo en alguna de las fases puede acarrear al tester volver a repetir la fase completa con sus activos, esto le supone una enorme presión al tester dado el coste temporal y de recursos que puede suponer un error (eso si se da cuenta y no le haya dado el visto bueno a un servicio habiendo quedado algo por testear).

5. Caso práctico real

Para completar el trabajo, se ha contextualizado el proceso del *testing* con un servicio real. Este servicio utiliza la tecnología framework 2 y consiste en la creación, modificación y borrado de una tabla de base de datos que corresponde a una tienda.

Durante el caso se muestran todos los datos del servicio que han resultado útiles para el tester y los pasos que se han seguido según el estándar de factoría (este caso ha sido probado e implementado con éxito en un entorno real, por lo tanto, tanto la calidad del servicio como la del proceso del *testing* ha sido confirmada).

5.1 Documento técnico de alto nivel (DTAN) [definido en la sección 4.2]

Como siempre ha de comenzar el *testing*, el tester ha de analizar el documento proporcionado por el cliente que indica las características del servicio.

5.1.1. Historial

Lo primero en lo que hay que fijarse es en el historial de cambios del servicio.

Historial de Cambios				
Versión	ID Petición	Fecha	Descripción	Responsable modificación
1.0	PM56913	10/02/2017	Nuevas operaciones CrearConfiguracionSurtidoGrupoCompra, BorrarConfiguracionSurtidoGrupoCompra y ModificarConfiguraciobSurtidoGrupoCompra	María Jesús Hernández (Capgemini)

Figura 48. *historial de cambios del caso real*

En la *figura 48* se puede visualizar que el servicio está en la versión 1.0 y no hay versiones anteriores, por lo tanto, es un nuevo desarrollo y habrá que testear todas sus operaciones (en la descripción nos menciona las 3 operaciones que tiene el servicio) completamente (es decir, no tenemos activos anteriores del servicio con los que partir).

Para no hacer el caso real más largo innecesariamente, me he centrado en la primera operación dado que la operación de creación es la que da más juego en el *testing* y donde suelen surgir más complicaciones para el tester.

5.1.2. Características del servicio

Una vez que se conoce el estado en el que se encuentra el servicio, hay que saber sus características (véase en la *figura 49*).

1.2 Características del Servicio

Aplicación	LOGIS
Nombre	ED_ConfiguracionSurtidoGrupoCompra_v1_0_0
Descripción	Operaciones simples para la configuración del surtido con el grupo de compra, tercero y localización origen que sirve el surtido
Tecnología de implementación	Framework 2
Perfil seguridad	K
Acción	Alta
Visibilidad	Público
Acceso	Intranet
Protocolo de Transporte	HTTP

Figura 49. Características del servicio del caso real

En esta sección hay que fijarse que todos los datos estén puestos. Lo más relevante para el *testing* es conocer la tecnología que emplea (en este caso es Framework 2).

El tester también ha de saber que tanto el nombre de la aplicación como el del servicio existan y, que el resto de los datos sean correctos según sus campos (esto se debe saber según los estándares que se usen o las instrucciones que haya dado el cliente durante el preproceso del *testing* (sección 4.1), de no ser así, hay que resolver la duda).

Características de la operación

Una vez conocido el servicio, el tester deberá conocer la operación (véase un ejemplo en la *figura 50*) donde ha de asegurarse que no falta ningún dato.

1.4.1 Operación CrearConfiguracionSurtidoGrupoCompra	
Nombre	CrearConfiguracionSurtidoGrupoCompra
Descripción	Operación que realiza el guardado en base de datos de la configuración del Producto, GC y tercero y localización origen
Acción	Alta
Criticidad	Media
Es transaccional	Sí
Excepciones transaccionalidad	No aplica Nota: Esta operación será llamada por otro servicio operacional que será el que tenga la transaccionalidad. Si no fuese llamado por otro servicio, la operación se comportará de modo transaccional
Requiere paginación	No
Registros por página	N/A
Requiere ordenación	No
Criterio de ordenación	N/A
Incluye adjuntos	No
Llamadas a otros servicios	• Sí • Llamadas locales al servicio (según aplique) : OP_GestionarAuditoria_v1_0_0
Errores Catálogo Factoría	ErrorNoControlado, ErrorFormatoParametrosEntrada
Errores Específicos	ErrorBBDDAuditoria
Particularidades	N/A

Figura 50. *Características de la operación del caso real*

Desde el punto de vista del tester, los datos más relevantes son:

- El nombre la **operación y su descripción** (hay que entender qué hace la operación).
- Si la operación es **transaccional** o no (si es transaccional, cuando salte algún error en mitad de la ejecución, todo lo que se ha hecho habrá de deshacerse).
- Si la operación usa la **paginación y la ordenación**. Esto es importante para el tester cuando se ejecuten los casos funcionales, ya que debe de saber si el resultado de la ejecución ha de seguir algún tipo de orden o no.
- **Llamadas a otros servicios**. El tester tendrá que saber si el servicio llama a otro para asegurarse de que esté desplegado durante la ejecución.

- Finalmente, el último dato relevante de esta sección son los **errores**. El tester debe asegurarse de comprobar que esos errores ocurran cuando se testee la aplicación y que no tenga ningún error más durante ésta.

5.1.3. Parámetros

Ahora que el tester conoce la operación, debe asegurarse de que todos los parámetros de entrada tengan todos los datos y sean coherentes entre ellos.

Parámetros de entrada						
Nombre	**Descripción**	**Tipo dato**	**Obligatorio**	**Valor defecto**	**Formato**	**Rango/ Valores posibles**
localeLanguage	I18N	String	Si	es	N/A	N/A
localeCountry	I18N	String	Si	ES	N/A	N/A
localeVariant	I18N	String	No	N/A	N/A	N/A
codUsuario	Código del usuario conectado en la aplicación cliente que invoca	String	Si	N/A	N/A	Longitud máx = 40
codAplicacion	Código de la aplicación que invoca a la operación	String	No	N/A	N/A	Longitud máx = 100
listaCrear	Lista de creación de datos	List	Si	N/A	N/A	[1…5]
[listaCrear] listaCrear data	Mapa con los datos	Map	Si	N/A	N/A	N/A
[listaCrear] [listaCrear data] codItem	Código del item	Long	Si	N/A	N/A	[0…99999 9999999]
[listaCrear] [listaCrear data] codGrupoCompra	Código del grupo de compra	Long	Si	N/A	N/A	[0…99999 9999999]
listaCrearTercero	Lista de creación de datos de proveedor	List	Si	N/A	N/A	[1…3]
[listaCrearTercero] listaCrearTercero data	Mapa con los datos de proveedor	Map	Si	N/A	N/A	N/A
[listaCrear] [listaCrear data] [listaCrearTercero] [listaCrearTercero data] codTerceroOrigen	Código del tercero Origen	Long	Si	N/A	N/A	[0…99999 9999999]
[listaCrear] [listaCrear data] codLocOrigen	Código de la localizacióon Origen	Long	Si	N/A	N/A	[0…99999 9999999]
[listaCrear] [listaCrear data] fechaInicioRds	Código de la fecha de inicio de la Red de suministro	Date	Si	N/A	N/A	N/A
[listaCrear] [listaCrear data] fechaInicio	Código de la fecha de inicio	Date	Si	N/A	N/A	N/A
[listaCrear] [listaCrear data] fechaFin	Código de la fecha fin	Date	Si	N/A	N/A	N/A

Parámetros de salida				
Nombre	**Descripción**	**Tipo de dato**	**Formato**	**Rango**
codError	Código del error	String	N/A	N/A
desError	Descripción del error	String	N/A	N/A

Figura 51. *Parámetros del caso real.*

En la *figura 51* es necesario destacar dos puntos muy importantes:

> **Cada lista ha de tener un mapa**, esto ayuda a los programadores a organizar los datos en determinados casos (cuando hay listas internas, paginación, llamadas desde otros servicios).
>
> **El rango del parámetro** no solo indica los valores que puede tener, sino también el máximo número de dígitos que permite.

5.1.4. Fechas

Este tipo de parámetro suele dar muchos problemas y confusiones, ya que el cliente puede querer una fecha muy concreta (incluyendo los milisegundos) o simplemente una fecha básica (en ciertos casos, como en la fecha de caducidad de un producto o la entrega de un paquete, puede ser algo muy importante).

Tratamiento de Fechas

Nombre	Parámetro / Columna	Truncar hasta precisión	Puede contener huso horario (S/N)
fechalnicioRds	Parámetro	Fecha	N
fechalnicio	Parámetro	Fecha	N
fechaFin	Parámetro	Fecha	N
FEC_D_CREACION	Columna	Fecha + Hora	N

Los posibles ***valores de la columna "Truncar hasta precisión"*** *serían:*

Tratamiento	*Precisión*
Fecha	*yyyy-mm-dd*
Fecha+Hora	*yyyy-mm-dd hh24:mi:ss*

Figura 52. *Tratamiento de fechas del caso real*

Como se muestra en la *figura 52*, el tester puede ver que se contemplan los tres parámetros del tipo fecha y el formato que ha de tener cada uno. (tener un formato definido en el estándar como se visualiza en la parte inferior de la *figura 52,* ayuda enormemente a evitar este tipo de problemas).

5.1.5. Cuadro de rendimiento

Este cuadro es importantísimo y esencial para la fase de rendimiento. La razón de esta afirmación se debe a que el cuadro tiene todos los datos que debe conocer el tester para poder realizar las pruebas de rendimiento del servicio, cualquier dato que no se haya especificado llevaría irremediablemente al bloqueo del proceso del *testing* hasta que se resuelva la duda.

Consideraciones para Rendimiento			
CONFIGURACIÓN DE LA PRUEBA	Usuarios concurrentes	Número medio	2
		Número máximo	3
	Frecuencia de uso		Se ejecutan 10 peticiones al día repartidas durante las 8:00 a 22:00h
	Escenario de pruebas		☐ E1 - Acciones que realizan todos los Usuarios disponiendo de menos de 4 horas. Por ejemplo: pedidos que deben realizarse entre las 8:00 y las 9:00 ☒ E2 - Acciones que realizan todos los Usuarios disponiendo entre 4 y 12 horas al día ☐ E3 - Acciones que realizan todos los Usuarios varias veces a lo largo de la semana *Acciones = invocaciones por operación*
	Excepciones al estándar para la prueba de Carga		N/A
CAMINO LÓGICO A RENDIR	Especificación del comportamiento		Camino lógico determinado por la secuencia de todos los pasos de los REQ-FUN-0001 (comportamiento principal)
	Parámetros de entrada no obligatorios a informar		N/A
	Criterio de ordenación		N/A
	¿Datos en preproducción?		No

Figura 53. *Consideraciones del rendimiento del caso real*

En la *figura 53* se encuentra un dato imprescindible:

- **El número de usuarios concurrentes** (se necesita un servidor que soporte esos usuarios durante el *testing* de rendimiento).

Además, hay dos datos muy relevantes que le proporciona una idea al tester sobre la ejecución de rendimiento (una operación que se ejecuta cada 10 minutos no puede tardar 1 minuto, pero una operación que se ejecuta 1 vez al día por un único usuario puede hacer que el cliente acepte el tiempo, aunque sea más tiempo de lo acordado inicialmente en el preproceso del *testing*).

- **El escenario de pruebas y la frecuencia de uso indican la criticidad de la operación**.

5.1.6. Diagrama de actividad

Una vez vistos los apartados anteriores, el tester podrá ser capaz de comprender lo que hace la operación y cómo se comporta. Para saber esto, tendrá un diagrama del comportamiento del servicio para tener una idea general y luego tendrá una sección donde se indique de forma más detallada el comportamiento.

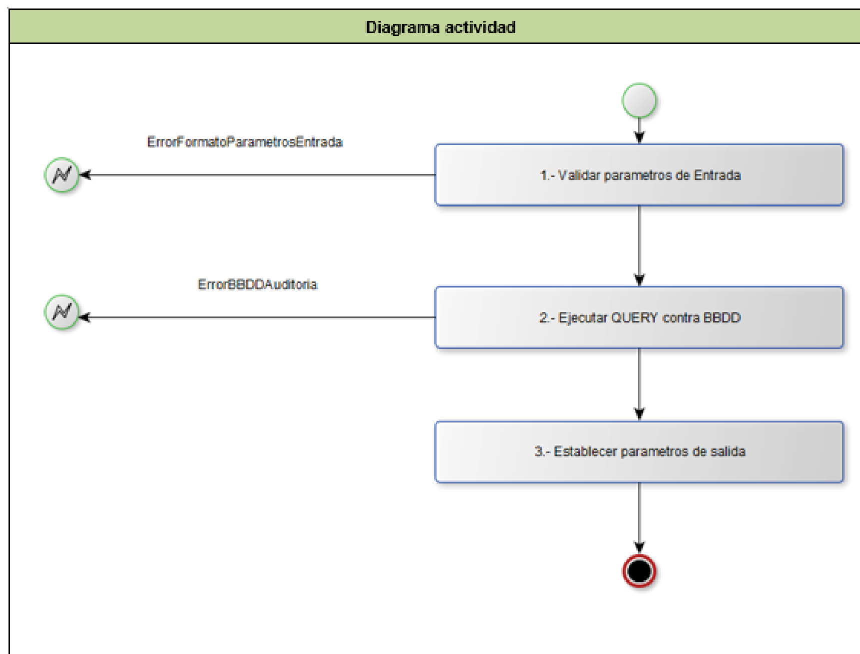


Figura 54. *Diagrama de actividad del caso real*

En la *figura 54,* el tester observa los tres pasos principales que tiene el servicio, los errores que tiene y donde éstos pueden surgir.

Hay que destacar que el **errorNoControlado** no aparece en el diagrama porque no hay forma de ubicarlo, es decir, puede saltar ese error en cualquiera de los pasos (aunque este error no aparece en el diagrama, siempre ha de mencionarse en el comportamiento ya que el error existe y puede aparecer durante su funcionamiento).

5.1.7. Comportamiento de la operación

Finalmente, el último apartado importante con respecto a este documento, es el comportamiento de la operación (se muestra un ejemplo en la *figura 55*), en este apartado se deberá indicar de forma específica y concreta qué hará la operación.

<table>
<tr><th colspan="2">REQ-FUN-0001 - Comportamiento principal</th></tr>
<tr><td>Descripción</td><td>Realiza creación de nuevos registros a partir de los parámetros de entrada</td></tr>
<tr><td>Paso 0</td><td>Si durante la ejecución de cualquier paso se produce un error no controlado terminar la operación con ErrorNoControlado.</td></tr>
<tr><td>Paso 1</td><td>Se crea una traza de inicio de operación con nivel INFO con el formato indicado en el apartado Anexos, 1.7

Validar parámetros de entrada según se indica en los apartados "Parametros de Entrada" y "Validaciones de entrada". Si se produce un error en la validación indicada en el apartado de validaciones de entrada, terminar la operación con ErrorFormatoParametrosEntrada.</td></tr>
<tr><td>Paso 2</td><td>Recorrer los elementos de la lista listaCrear y realizar la inserción de los datos en la tabla:
<table>
<tr><th>Tabla</th><th>Alias</th></tr>
<tr><td>S_CONF_SURTIDO_GC</td><td></td></tr>
</table>
Valores de los campos a insertar:
<table>
<tr><th>Campo</th><th>Parámetro Entrada/Valor</th></tr>
<tr><td>COD_N_ITEM</td><td>[listaCrear][listaCrear data] codItem</td></tr>
<tr><td>COD_N_GRUPO_COMPRA</td><td>[listaCrear][listaCrear data] codGrupoCompra</td></tr>
<tr><td>COD_N_TERCERO_ORIGEN</td><td>[listaCrear] [listaCrear data] [listaCrearTercero] [listaCrearTercero data] codTerceroOrigen</td></tr>
<tr><td>COD_N_LOCALIZACION_ORIGEN</td><td>[listaCrear][listaCrear data] codLocOrigen</td></tr>
<tr><td>FEC_D_FECHA_INICIO_RDS</td><td>[listaCrear][listaCrear data] fechaInicioRds</td></tr>
<tr><td>FEC_D_FECHA_INICIO</td><td>[listaCrear][listaCrear data] fechaInicio</td></tr>
<tr><td>FEC_D_FECHA_FIN</td><td>[listaCrear][listaCrear data] fechaFin</td></tr>
<tr><td>COD_V_USR_CREACION</td><td>codUsuario</td></tr>
<tr><td>FEC_D_CREACION</td><td>Fecha y hora del sistema cuando se realizó la operación</td></tr>
</table>
Se crea una traza de paso de la operación con nivel INFO con el formato indicado en el apartado Anexos, 1.7

En caso de error de BBDD lanzar el error ErrorBBDDAuditoria .</td></tr>
<tr><td>Paso 3</td><td>Si la operación termina con éxito, devolver el resultado de la Ejecución:

Establecer parametros de salida:
<table>
<tr><th>Parámetro Salida</th><th>Valor</th></tr>
<tr><td>codError</td><td>Null</td></tr>
<tr><td>desError</td><td>Null</td></tr>
</table>
Se crea una traza de fin de operación con nivel INFO con el formato indicado en el apartado Anexos,</td></tr>
</table>

Figura 55. *Comportamiento principal del caso real*

A continuación, explicaré los pasos que sigue el servicio además de los datos importantes que se mencionan y que el tester ha de tener en cuenta para el *testing*.

- **Paso 0**
  - La operación contempla el errorNoControlado.
- **Paso 1**
  - La operación valida los parámetros de entrada y si hay cualquier problema, salta el error que le corresponde.
- **Paso 2**
  - Aparece la tabla que usa la operación y las columnas a las que les asociará un valor de los parámetros de entrada. Finalmente contempla el error de base de datos por si hay algún problema al insertar los datos debido a un error por clave ajena o por incompatibilidad entre la columna y el parámetro (por ejemplo, tratar de asignarle un valor de tipo string a una columna de tipo Number).

- **Paso 3**
    - Indica el valor de los parámetros de salida.

Hay que mencionar que en el comportamiento también se indican las trazas que escribirá la operación, éstas ayudarán al tester a localizar algún fallo que pueda tener y registrará en logs lo que ha ocurrido en la ejecución (algo muy útil cuando varios servicios se ejecutan al mismo tiempo llamándose entre ellos).

5.2. Documento técnico detallado (DTD) [Definido en la sección 4.3]

Una vez revisado el DTAN y se haya familiarizado el tester con el servicio, puede disponerse a analizar en mayor profundidad el servicio revisando las clases y métodos que lo conforman. Este proceso de validacion y revisión comenzará con la operación a testear.

5.2.1. Operación

El documento debe tener la operación, indicando: su tipo, la tabla que usa y las columnas que modifica.

Nombre:	logis.crearConfSurtidoGC
Tipo:	INSERT

Descripción:	`INSERT ALL`
	`<iterate property="listaCrear">`
	`<iterate property="listaCrear[].listaCrearTercero">`
	`INTO`
	`S_CONF_SURTIDO_GC`
	`(COD_N_ITEM,`
	`COD_N_GRUPO_COMPRA,`
	`COD_N_TERCERO_ORIGEN,`
	`COD_N_LOCALIZACION_ORIGEN,`
	`FEC_D_FECHA_INICIO_RDS,`
	`FEC_D_FECHA_INICIO,`
	`FEC_D_FECHA_FIN,`
	`COD_V_USR_CREACION,`
	`FEC_D_CREACION)`
	`VALUES (`
	`#listaCrear[].codItem:DECIMAL#,`
	`#listaCrear[].codGrupoCompra:DECIMAL#,`
	`#listaCrear[].listaCrearTercero[].codTerceroOrigen:DECIMAL#,`
	`#listaCrear[].codLocOrigen:DECIMAL#,`
	`TRUNC(#listaCrear[].fechaInicioRds:DATE#),`
	`TRUNC(#listaCrear[].fechaInicio:DATE#),`
	`TRUNC(#listaCrear[].fechaFin:DATE#),`
	`#codUsuario:VARCHAR#,`
	`SYSDATE)`
	`</iterate>`
	`</iterate>`
	`SELECT * FROM DUAL`

Id secuencia:	N/A
Tablas involucradas:	S_CONF_SURTIDO_GC
Mapping de entrada:	N/A
Mapping de salida:	N/A

Figura 56. Operación del DTD. Caso real

En la *figura 56* se puede observar que las fechas se truncan para no tener datos innecesarios como los milisegundos (esto se consigue con la instrucción "TRUNC").

5.2.2. Errores de la operación

La segunda sección más importantes es esta (véase un ejemplo en la *figura 57*), donde se mostrarán los errores de la operación, indicando principalmente: su nombre, descripción, y los mensajes que se registrarán en la traza y que visualizará el usuario (mensaje amigable).

4.2.1 ErrorFormatoParametrosEntrada

Nombre:	ErrorFormatoParametrosEntrada
Descripción:	Alguno de los parámetros de entrada no cumple su formato
CodigoError:	logis.common.error.formatoParametrosEntrada
Mensaje Amigable:	Parámetros de entrada con formato incorrecto. [Parámetros: {0}] 1. {0}: Parámetros (separado por comas) de todos los parámetros de entrada que no cumplen su formato (ej " codItem, codLocalizacionServicio, codGrupoCompra, codTienda, codTipoServicio, codTipoVariableLogistica").
Mensaje Trazado:	{0} – {1} – {2} 1. {0}: Nombre del servicio web. 2. {1}: Nombre de la operación del servicio web. 3. {2}: Mensaje descrito en el apartado Mensaje Amigable.

4.2.2 ErrorNoControlado

Nombre:	ErrorNoControlado
Descripción:	Error no controlado durante la ejecución de la operación.
CodigoError:	logis.common.error.noControlado
Mensaje Amigable:	Error no controlado.
Mensaje Trazado:	{0} – {1} – Error no controlado: [{2}] 1. {0}: Nombre del servicio web. 2. {1}: Nombre de la operación del servicio web. 3. {2}: Pila de la excepción de base de datos que provoca el error.

4.2.3 ErrorBBDDAuditoria

Nombre:	ErrorBBDDAuditoria
Descripción:	Error al realizar una operación sobre una base de datos.
CodigoError:	logis.common.error.bbdd.auditoria
Mensaje Amigable:	{0} – {1} – Error ejecutando operación sobre base de datos 1. {0}: Nombre del servicio web. 2. {1}: Nombre de la operación del servicio web. Ejemplo: *ED_FormatoPedido* – *ConsultarFormatoPedido* – Error ejecutando operación sobre base de datos
Mensaje Trazado:	{0} – {1} – Error ejecutando operación sobre base de datos{2} 1. {0}: Nombre del servicio web. 2. {1}: Nombre de la operación del servicio web. 3. {2}: Pila de la excepción de base de datos que provoca el error.

Figura 57. *Características de los errores del DTD. Caso real*

5.3. Fase de pruebas funcionales [Definido en la sección 4.4]

Una vez el tester ha validado todos los documentos que le ha entregado el cliente, el tester puede disponerse a realizar la fase funcional.

A continuación, se detalla la fase funcional con sus activos y su correspondiente ejecución.

5.3.1. Plan de pruebas (PLP) [Definido en la sección 4.4.1]

Una vez validados los documentos proporcionados por el cliente, el tester puede empezar a realizar los activos necesarios para testear el servicio. Este proceso ha de comenzar con los casos de pruebas del activo PLP.

5.3.1.1. Casos de Pruebas

El activo debe contemplar todas las validaciones de la operación. Una vez comprobados, también hay que hacer unos casos al final que confirmen todos los errores de la operación (así como comprobar que no tiene ningún otro error).

REQUERIMIENT	TC ID	Objetivo	Descripción	
REQ-FUN-0001	OP_01-F-001	SC	Funcional	Se da de alta un único registro usando sólo los parámetros obligatorios.
REQ-FUN-0001	OP_01-F-002	----	Funcional	Se dan de alta dos o más registros.
REQ-FUN-0001	OP_01-F-003	----	Funcional	Se da de alta un único registro usando todos los parámetros de entrada.
REQ-FUN-0001	OP_01-F-004	----	Validación	Se informa en el límite inferior los parámetros de tipo integer/double/long. Se informa con la longitud mínima el parámetro de tipo string/char.
REQ-FUN-0001	OP_01-F-005	----	Validación	Se informa en el límite superior los parámetros de tipo integer/double/long. Se informa con la longitud máxima el parámetro de tipo string/char codUsuario
REQ-FUN-0001	OP_01-F-006	----	Funcional	Se dan de alta dos o más registros con todos los parametros informados
REQ-FUN->E001	OP_01-F-007	----	Funcional	Se informa con cadena vacía: • localeLanguage • localeCountry • localeVariant • codUsuario • codAplicacion
REQ-FUN->E001	OP_01-F-008	----	Funcional	Se informa con espacios en blanco: • localeLanguage • localeCountry • localeVariant • codUsuario • codAplicacion
REQ-FUN-E001	OP_01-F-009	----	Validación - Erroneo	Se informa con una longitud superior a la máxima el parámetro de tipo string/char 'codUsuario' y el parametro "codAplicacion"
REQ-FUN-E001	OP_01-F-010	----	Validación - Erroneo	Se informa por debajo del límite inferior el parámetro de tipo integer/double
REQ-FUN-E001	OP_01-F-011	----	Validación - Erroneo	Se informa por encima del límite inferior el parámetro de tipo integer/double
REQ-FUN-E002	OP_01-F-012	----	Funcional - Erroneo	Se intenta dar de alta un registro cuyo parámetro COD_N_TERCERO (clave ajena) no existe en la tabla D_TERCERO_R
REQ-FUN-E002	OP_01-F-013	----	Funcional - Erroneo	Se intenta dar de alta un registro cuyo parámetro COD_N_LOCALIZACION (clave ajena) no existe en la tabla D_LOCALIZACION_R.
REQ-FUN-E002	OP_01-F-014	----	Funcional - Erroneo	Se intenta dar de alta un registro cuyo parámetro COD_N_ITEM; COD_N_GRUPO_COMPRA) no existe en la tabla S_RDS_SURTIDO_GC.
REQ-FUN-E002	OP_01-F-015	----	Funcional - Erroneo	Se intenta dar de alta un registro cuya clave primaria ya existe en BBDD.
REQ-FUN-E002	OP_01-F-016	----	Funcional - Erroneo	Error de base de datos.
REQ-FUN-E003	OP_01-F-017	----	Funcional - Erroneo	Error no controlado.Consulta Paso 4 REQ-FUN-001 no devuelve ningun registro.
REQ-FUN-E001	OP_01-F-018	----	Validación - Erroneo	Se informa con cadena vacia 'codUsuario"
REQ-FUN-E001	OP_01-F-019	----	Validación - Erroneo	Se informa con espacios en blanco 'codUsuario"

Figura 58. *Casos de prueba del PLP. Caso real*

En la *figura 58* hay que destacar que es necesario que se realicen casos con valores de espacios en blanco o cadena vacía. Esto se debe a que algunos lenguajes de programación tratan estos valores de forma diferente a valores con caracteres (los tratan como valores nulos), por lo tanto, siempre han de estar estos valores.

Siempre ha de haber casos que confirmen los rangos de los parámetros, así como todas las validaciones que, según el DTAN, realiza el servicio.

Para finalizar, es esencial el caso dos, donde se crea más de un registro. Esto se debe a que cuando hay más de un registro, hay que asegurarse que se crean todos los registros y en el orden que toca, además, si hubiera ordenación habría que detallar los valores de cada registro para asegurar al tester que lo aplica (en este caso, al no haber ordenación no es necesario).

5.3.2. Entrada funcional (ETF) [Definido en la sección 4.4.2]

Una vez realizado el PLP, el tester debe hacer el ETF. El activo debe crear una tabla con tantas columnas como parámetros de entrada. Además, tendrá tantas inserciones como registros de entrada tiene el PLP.

```
1  -------------------------------------
2  -- OP_01 -> Operacion de creacion --
3  -------------------------------------
4
5  --------------------------
6  -- Borrado de datos
7  --------------------------
8  DELETE FROM LOGIS_ED_ConfiguracionSurtidoGrupoCompra_OP01;
9  COMMIT;
10 --------------------------
11 -- Borrado de tabla
12 --------------------------
13 DROP TABLE LOGIS_ED_ConfiguracionSurtidoGrupoCompra_OP01;
14 COMMIT;
15 --------------------------
16 -- Creación de la tabla
17 --------------------------
18 CREATE TABLE LOGIS_ED_ConfiguracionSurtidoGrupoCompra_OP01 (
19     TC_ID VARCHAR2(200 BYTE) NOT NULL ENABLE,
20     LOCALELANGUAGE VARCHAR2(2000 BYTE),
21     LOCALECOUNTRY VARCHAR2(2000 BYTE),
22     LOCALEVARIANT VARCHAR2(2000 BYTE),
23     CODUSUARIO VARCHAR2(2000 BYTE),
24     CODAPLICACION VARCHAR2(2000 BYTE),
25     CODITEM VARCHAR2(2000 BYTE),
26     CODGRUPOCOMPRA VARCHAR2(2000 BYTE),
27     CODTERCEROORIGEN VARCHAR2(2000 BYTE),
28     CODLOCORIGEN VARCHAR2(2000 BYTE),
29     FECHAINICIORDS VARCHAR2(2000 BYTE),
30     FECHAINICIO VARCHAR2(2000 BYTE),
31     FECHAFIN VARCHAR2(2000 BYTE),
32     DEFAULT '[ver PLP]'
33 );
34 COMMIT;
```

Figura 59. *Tabla del activo ETF del caso real*

En la *figura 59* se muestra lo mencionado anteriormente con un borrado de datos y de tabla por si el tester desea eliminarlo (suele hacerse estos “deletes” por defecto por si se modifican los parámetros de entrada o el activo PLP).

Figura 60. *Inserciones del activo ETF del caso real*

En la *figura 60* se observa un ejemplo que corresponde con los primeros ejemplos del activo PLP, cada línea corresponde a un registro de la tabla con unos datos para cada parámetro que deben coincidir con los indicados en el PLP (anteriormente explicado).

5.3.3. Datos técnicos (DAT) [Definido en la sección 4.4.4]

Una vez definido el ETF, el siguiente paso del tester consiste en la realización del DAT para llenar las tablas de la base de datos que el servicio use directa o indirectamente (como se ha mencionado al principio del caso real, este documento es necesario porque la operación influye sobre la base de datos).

El activo (se muestra un ejemplo en la *figura 61*) debe realizar las inserciones pertinentes para que pueda realizar la inserción, esto ocurre debido a que la tabla que utiliza el servicio tiene diversas claves ajenas con otras tablas, por lo tanto, para que no salte un error de clave ajena al insertar en la base de datos, hay que hacer estas inserciones antes de ejecutar el servicio.

Figura 61. *Inserciones del activo DAT. Caso real*

Además de lo que se ha mencionado anteriormente, la última inserción que se encuentra comentada es la que debería hacer el servicio si éste funciona correctamente.

Es recomendable hacer la inserción manualmente (luego eliminar el registro antes de ejecutar el servicio para que no salte error por registro duplicado) para garantizar de que el tester no ha pasado nada por alto (es la única validación y revisión del documento que se realiza antes de empezar a ejecutar los casos).

5.3.4. WSDL [Definido en la sección 4.4.6]

Una vez realizado el PLP, deberá crearse el WSDL (véase un ejemplo en la *figura 62*) para poder preparar el SCF en el *soapUI* (activo explicado en la sección 4).

```
<xsd:complexType name="Input_CrearConfiguracionSurtidoGrupoCompra">
  <xsd:complexContent>
    <xsd:extension base="ns0:WebServicesInput">
      <xsd:sequence>
        <xsd:element minOccurs="0" name="codAplicacion" nillable="true" type="xsd:string"/>
        <xsd:element name="codUsuario" type="xsd:string"/>
        <xsd:element minOccurs="0" name="listaCrear" nillable="true" type="tns:ArrayOfListaCrearData-0-"/>
        <xsd:element name="localeCountry" type="xsd:string"/>
        <xsd:element name="localeLanguage" type="xsd:string"/>
        <xsd:element minOccurs="0" name="localeVariant" nillable="true" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="listaCrearData">
  <xsd:sequence>
    <xsd:element name="codGrupoCompra" type="xsd:long"/>
    <xsd:element name="codItem" type="xsd:long"/>
    <xsd:element name="codLocOrigen" type="xsd:long"/>
    <xsd:element name="fechaFin" type="xsd:dateTime"/>
    <xsd:element name="fechaInicio" type="xsd:dateTime"/>
    <xsd:element name="fechaInicioRds" type="xsd:dateTime"/>
    <xsd:element minOccurs="0" name="listaCrearTercero" nillable="true" type="tns:ArrayOfListaCrearTerceroData-0-"/>
  </xsd:sequence>
</xsd:complexType>
```

Figura 62. *Entrada del WSDL. Caso real*

En el activo deben estar los parámetros de entrada y cada uno debe tener, además del nombre: su tipo y si es “nillable” o no (es decir, si puede tener valor nulo).

Dado que una lista es más compleja que una variable simple, se debe tratar aparte de la entrada de la operación (debido a esto, en la entrada solo aparece el nombre de la tabla y los parámetros que contiene se definen en otra sección).

```
<xsd:complexType name="Output_CrearConfiguracionSurtidoGrupoCompra">
  <xsd:complexContent>
    <xsd:extension base="ns0:WebServicesOutput">
      <xsd:sequence>
        <xsd:element minOccurs="0" name="codError" nillable="true" type="xsd:string"/>
        <xsd:element minOccurs="0" name="desError" nillable="true" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figura 63. *Salida del WSDL. Caso real*

El documento también debe tener la salida (ejemplo mostrado en la *figura 63*) con sus correspondientes parámetros.

Dicha salida debe tener al menos dos parámetros: un parámetro que guarde el código del error (nulo si no hay ninguno) y otro que describa el error que ha ocurrido (nulo si no ha saltado ningún error).

5.3.5. Script funcional (SCF) [Definido en la sección 4.4.7]

Una vez realizados: el ETF, el DAT y el WSDL. El tester puede empezar a preparar el SCF.

El activo debe contener todas las operaciones y, para cada operación, debe tener todos los casos con sus correspondientes valores en los parámetros.

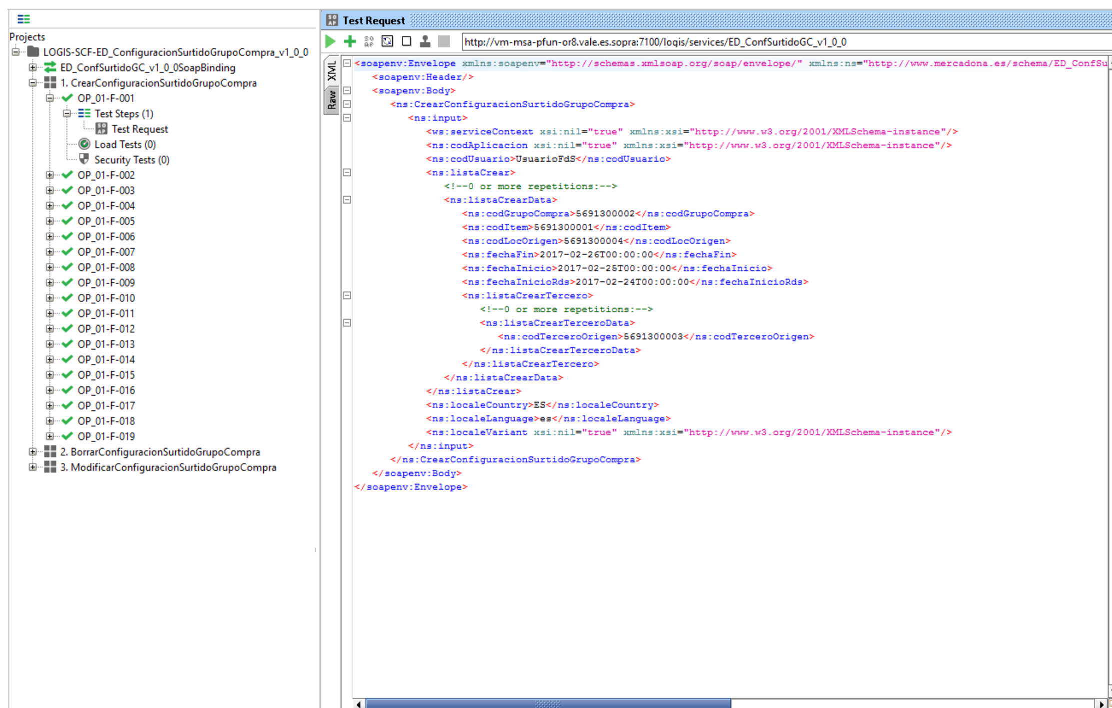


Figura 64. *Captura del SCF. Caso real*

En la *figura 64* se puede observar el script que le corresponde al servicio y los valores que posee el primer caso de la operación de creación.

En la parte superior está indicado el servidor donde se ejecutará el servicio y la versión del servicio que se ejecutará.

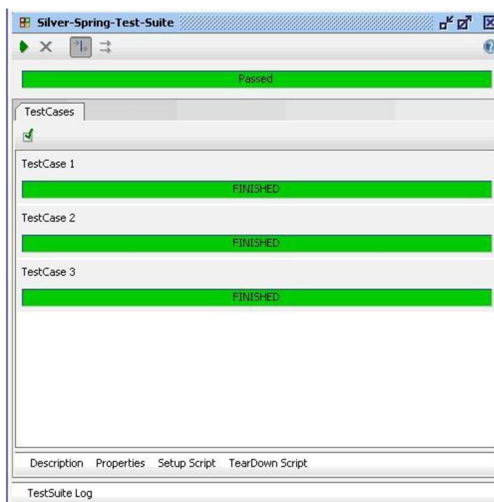


Figura 65. *Resultados de la ejecución funcional. Caso real*

Una vez preparado todos los activos funcionales y el tester haya podido dar un OK funcional (se muestra un ejemplo de "OK" en la *figura 65*), se dispondrá a realizar el *testing* del rendimiento.

Si surge algún problema o error durante la fase funcional, el tester tendrá que interrumpir el proceso del *testing* hasta que el tester (si es por algún activo mal realizado) o el cliente solucione los errores y le proporcione al tester un servicio completamente funcional (el tester tendría que repetir la fase funcional de los casos a los que pueda afectar los cambios aplicados al servicio).

5.4. Fase de pruebas de seguridad [Definido en la sección 4.5]

Una vez finalizada la fase funcional del *testing*, el tester tendrá que continuar con la fase de seguridad para comprobar si todos y cada uno de los parámetros del servicio son seguros tanto para el cliente como para el usuario.

5.4.1. Script de seguridad (SCS) [Definido en la sección 4.5.1]

Si todo ha ido correctamente en la fase funcional, el tester tendrá que continuar con la fase de seguridad y, para esto, empezará preparando un nuevo proyecto *SoapUI*.

El activo debe de tener todas las operaciones del servicio y, dentro de cada operación ha de contener el "test request" con todos los parámetros y los dos "test" de seguridad.

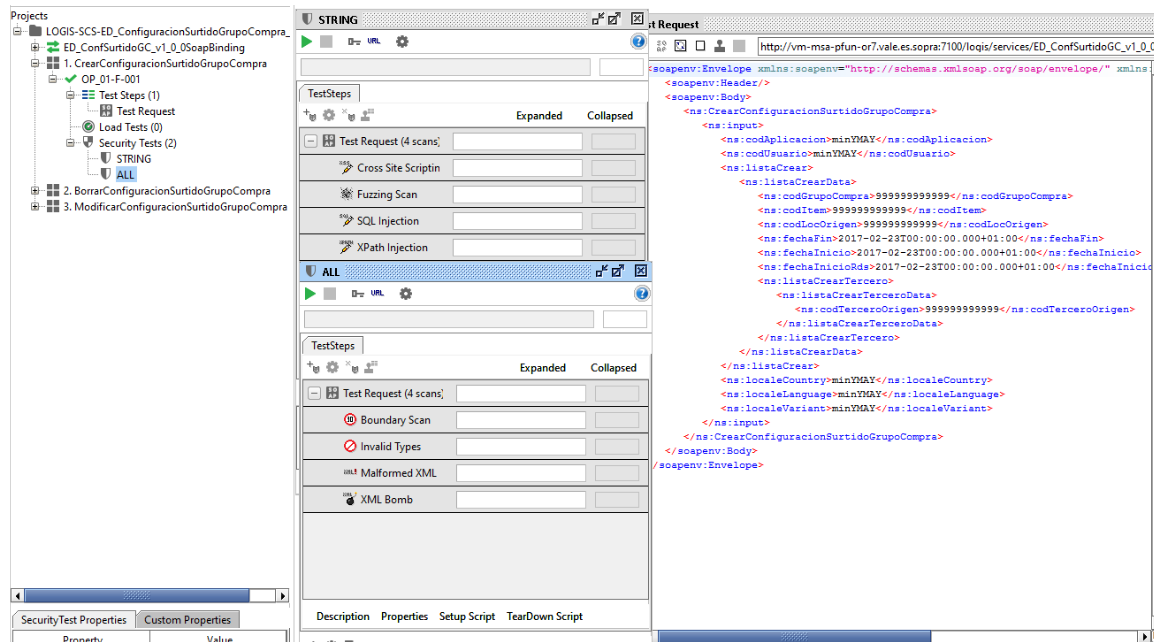


Figura 66. *Captura del SCS. Caso real*

La *figura 66* muestra el "test" de seguridad que comprueba todos los parámetros de entrada, donde:

- Los parámetros de tipo String se les han asignado un valor con minúsculas y mayúsculas.

- Las fechas tienen un valor que incluyen los milisegundos.
- Los valores de tipo numérico se les asocia el valor más alto que son capaces de soportar.

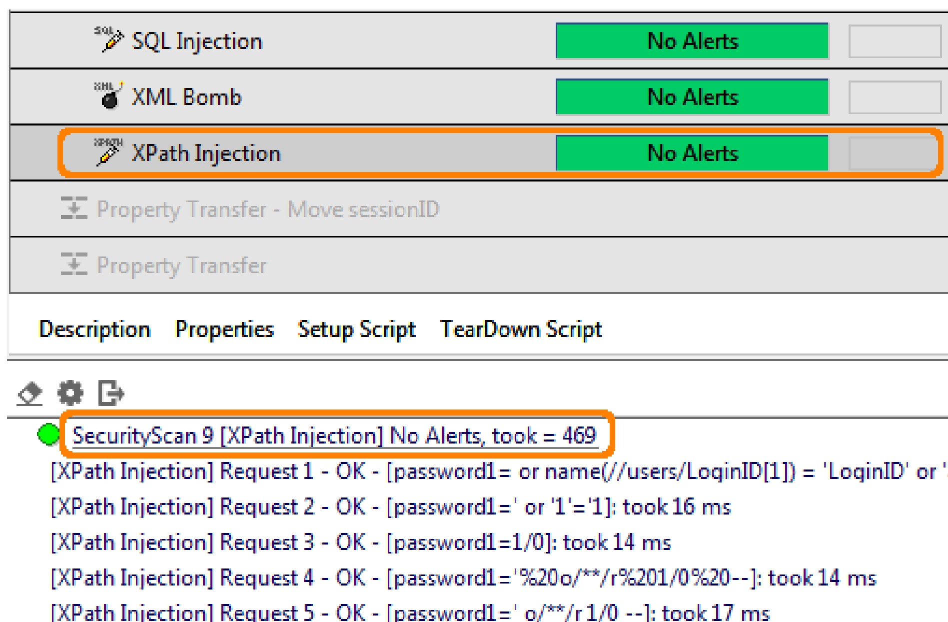


Figura 67. *Resultados del activo SCS. Caso real*

Si todo ha ido correctamente (se muestra un ejemplo de un resultado con éxito en la *figura 67*) y el tester ha dado un OK funcional y de seguridad, puede ir al último paso del *testing* y finalizar el proceso.

5.5. Fase de rendimiento [Definido en la sección 4.6]

La última fase del testing es la fase de rendimiento, el tester tendrá que realizar esta fase para comprobar si el servicio, además de cumplir con su funcionalidad y tener un buen nivel de seguridad, es de calidad y tiene buenos tiempos de respuesta.

A continuación, se detallan los activos necesarios para esta fase con su correspondiente ejecución.

5.5.1. Documento de alto rendimiento (DAR) [Definido en la sección 4.6.1]

El primer activo con el que ha de empezar el tester en la fase de rendimiento es el DAR, dicho activo deberá de tener tanto la tabla que usa el servicio como las tablas que se necesitan por claves ajenas.

```xml
<table>
    <name>D_LOCALIZACION_R</name>
    <columns>
        <column>
            <name>COD_N_LOCALIZACION</name>
            <type>Number</type>
            <init>5691300004</init>
            <incremental value="1">true</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
        <column>
            <name>COD_N_TIPO_LOCALIZACION</name>
            <type>Number</type>
            <init>10001</init>
            <incremental value="1">false</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
        <column>
            <name>TXT_NOMBRE</name>
            <type>varchar2</type>
            <init>Localizacion 1</init>
            <incremental value="1">true</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
    </columns>
    <nrows>100000</nrows>
</table>
```

```xml
<table>
    <name>D_TERCERO_R</name>
    <columns>
        <column>
            <name>COD_N_TERCERO</name>
            <type>Number</type>
            <init>5691300003</init>
            <incremental value="1">true</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
        <column>
            <name>COD_N_TIPO_TERCERO</name>
            <type>Number</type>
            <init>10002</init>
            <incremental value="1">false</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
    </columns>
    <nrows>500000</nrows>
</table>
```

```xml
<table>
    <name>D_ITEM_R</name>
    <columns>
        <column>
            <name>COD_N_ITEM</name>
            <type>Number</type>
            <init>5691300001</init>
            <incremental value="1">true</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
        <column>
            <name>COD_N_TIPO_ITEM</name>
            <type>Number</type>
            <init>10003</init>
            <incremental value="1">false</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
    </columns>
    <nrows>100000</nrows>
</table>
```

```xml
<table>
    <name>D_RDS_GRP_COMPRA</name>
    <columns>
        <column>
            <name>COD_N_GRUPO_COMPRA</name>
            <type>Number</type>
            <init>5691300002</init>
            <incremental value="1">true</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
        <column>
            <name>TXT_DESCRIPCION_CORTA</name>
            <type>varchar2</type>
            <init>Grupo Compra</init>
            <incremental value="1">true</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
    </columns>
    <nrows>182</nrows>
</table>
```

Figura 68. *Tablas del activo DAT. Caso real*

Estas tablas crearán cientos o miles de registros. Para poder hacer esto existen los parámetros “incremental” y “frequency¨ que automáticamente cambiarán el valor de cada uno de los registros de la tabla para que no haya repeticiones (por lo tanto, tampoco habrá errores por registro duplicado).

Hay que señalar que sólo se indican las columnas que usa el servicio y las que sean clave primaria de cada tabla, esto se debe a que no son necesarios cambiar los valores de las otras columnas para la creación de los registros.

```xml
<table>
    <name>S_RDS_SURTIDO_GC</name>
    <columns>
        <column>
            <name>COD_N_ITEM</name>
            <type>Number</type>
            <init>5691300001</init>
            <incremental value="1">true</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
        <column>
            <name>COD_N_GRUPO_COMPRA</name>
            <type>Number</type>
            <init>5691300002</init>
            <incremental value="1">true</incremental>
            <frequency>1</frequency>
            <minValue>5691300002</minValue>
            <maxValue>5691300181</maxValue>
        </column>
        <column>
            <name>FEC_D_FECHA_INICIO</name>
            <type>Sysdate</type>
            <ctrlValue>TRUNC(SYSDATE+:FEC_D_FECHA_INICIO)</ctrlValue>
            <init>0</init>
            <incremental value="1">false</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
        <column>
            <name>FEC_D_FECHA_FIN</name>
            <type>Sysdate</type>
            <ctrlValue>TRUNC(SYSDATE+:FEC_D_FECHA_INICIO)</ctrlValue>
            <init>0</init>
            <incremental value="1">false</incremental>
            <frequency>1</frequency>
            <minValue>null</minValue>
            <maxValue>null</maxValue>
        </column>
    </columns>
    <nrows>66500</nrows>
</table>
```

Figura 69. *Tabla del activo DAT. Caso real*

Finalmente, como se observa en la *figura* 69, se podrán generar miles de registros de la tabla que usa el servicio. Así el tester conocerá el comportamiento del servicio cuando trabaje con una tabla con muchos registros al mismo tiempo y tenga que comprobar si el registro a crear es un registro repetido o si existe espacio en la tabla para crearlo.

5.5.2. Entrada de Rendimiento (ETR) [Definido en la sección 4.6.2]

Al igual que el DAR sirve para dar miles de registros en la base de datos, el ETR sirve para asociar cientos o miles de valores a los parámetros (un valor para cada parámetro por cada una de las cientos o miles de ejecuciones del servicio).

El activo (véase un ejemplo en la *figura 70*) debe de tener todos los valores necesarios para cada uno de los parámetros, además de tener los suficientes valores para poder hacer cualquier test de rendimiento (explicados en el apartado 4.6).

	A	B	C	D	E	F	G	H	I	J
1	5691300001	5691300002	5691300003	5691300004	5691300005	5691300002	5691300003	5691300004	5691300005	5691300006
2	5691300006	5691300007	5691300008	5691300009	5691300010	5691300007	5691300008	5691300009	5691300010	5691300011
3	5691300011	5691300012	5691300013	5691300014	5691300015	5691300012	5691300013	5691300014	5691300015	5691300016
4	5691300016	5691300017	5691300018	5691300019	5691300020	5691300017	5691300018	5691300019	5691300020	5691300021
5	5691300021	5691300022	5691300023	5691300024	5691300025	5691300022	5691300023	5691300024	5691300025	5691300026
6	5691300026	5691300027	5691300028	5691300029	5691300030	5691300027	5691300028	5691300029	5691300030	5691300031
7	5691300031	5691300032	5691300033	5691300034	5691300035	5691300032	5691300033	5691300034	5691300035	5691300036
8	5691300036	5691300037	5691300038	5691300039	5691300040	5691300037	5691300038	5691300039	5691300040	5691300041
9	5691300041	5691300042	5691300043	5691300044	5691300045	5691300042	5691300043	5691300044	5691300045	5691300046
10	5691300046	5691300047	5691300048	5691300049	5691300050	5691300047	5691300048	5691300049	5691300050	5691300051
11	5691300051	5691300052	5691300053	5691300054	5691300055	5691300052	5691300053	5691300054	5691300055	5691300056
12	5691300056	5691300057	5691300058	5691300059	5691300060	5691300057	5691300058	5691300059	5691300060	5691300061
13	5691300061	5691300062	5691300063	5691300064	5691300065	5691300062	5691300063	5691300064	5691300065	5691300066
14	5691300066	5691300067	5691300068	5691300069	5691300070	5691300067	5691300068	5691300069	5691300070	5691300071
15	5691300071	5691300072	5691300073	5691300074	5691300075	5691300072	5691300073	5691300074	5691300075	5691300076
16	5691300076	5691300077	5691300078	5691300079	5691300080	5691300077	5691300078	5691300079	5691300080	5691300081
17	5691300081	5691300082	5691300083	5691300084	5691300085	5691300082	5691300083	5691300084	5691300085	5691300086
18	5691300086	5691300087	5691300088	5691300089	5691300090	5691300087	5691300088	5691300089	5691300090	5691300091

Figura 70. *Datos del activo DAT. Caso real*

En este caso, dado que se desea someter a una prueba de carga al servicio, se han creado multitud de datos para crear cientos o incluso miles de registros y saber la respuesta del servicio a una creación masiva de registros.

Es muy útil las herramientas que posee el Excel para facilitar la creación de estos valores de una forma rápida y óptima (como la secuencialidad).

5.5.3. Script de Rendimiento (SCR) [Definido en la sección 4.6.3]

Una vez que el tester posea el DAR y el ETR, puede empezar a preparar el SCR para ejecutar el *testing* de rendimiento.

El activo debe contener todas las operaciones del servicio y, para cada una de ellas, debe tener un caso de carga (test de rendimiento).

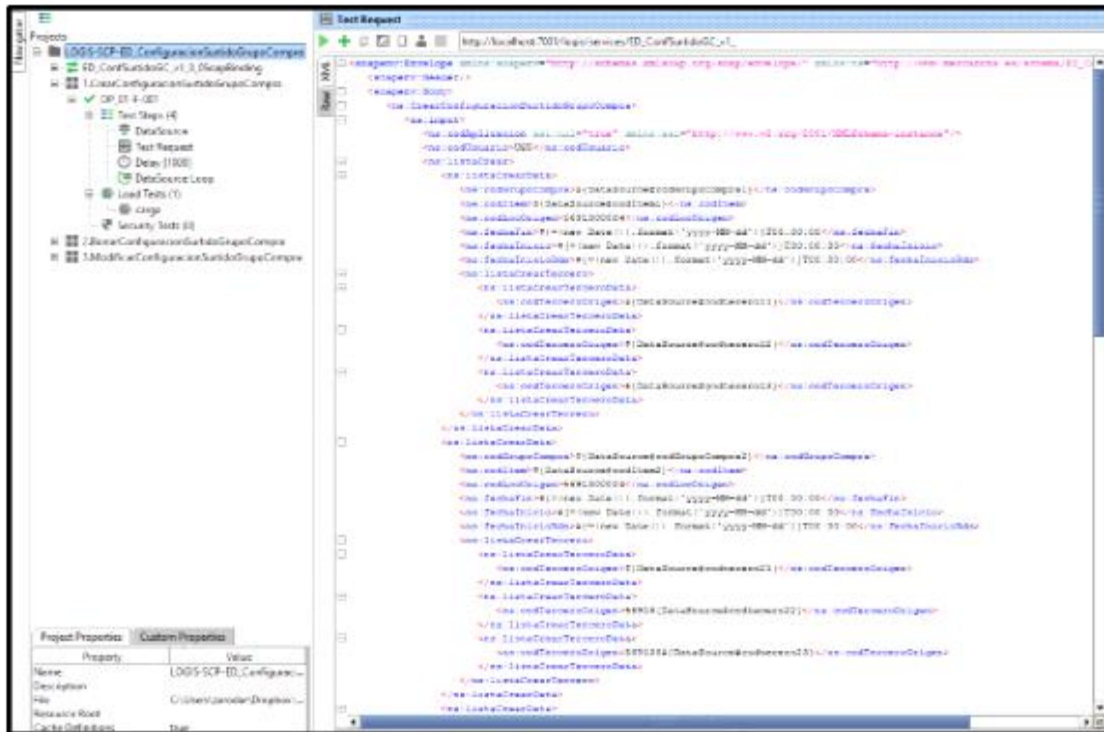


Figura 71*. Captura del SCR. Caso real*

En la *figura 71* se observa dos elementos esenciales en el SCR:

1. el **datasource** para seleccionar el activo ETR y asociar sus valores a los parámetros del "test request" ("test" que ejecutará los casos).
2. el elemento **delay** que obliga a hacer una pausa de 1 segundo entre las peticiones que se repetirán en bucle debido al datasource loop.

carga

Limit: 60 Seconds

Threads: 15 Strategy Simple Test Delay 1000 Random 0.5

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
DataSource	0	0	0	0	0	0	0	0	0	0
Test Request	0	0	0	0	0	0	0	0	0	0
Delay [1000]	0	0	0	0	0	0	0	0	0	0
DataSource Loop	0	0	0	0	0	0	0	0	0	0
TestCase:	0	0	0	0	0	0	0	0	0	0

Show Types: - All - Show Steps: - All -

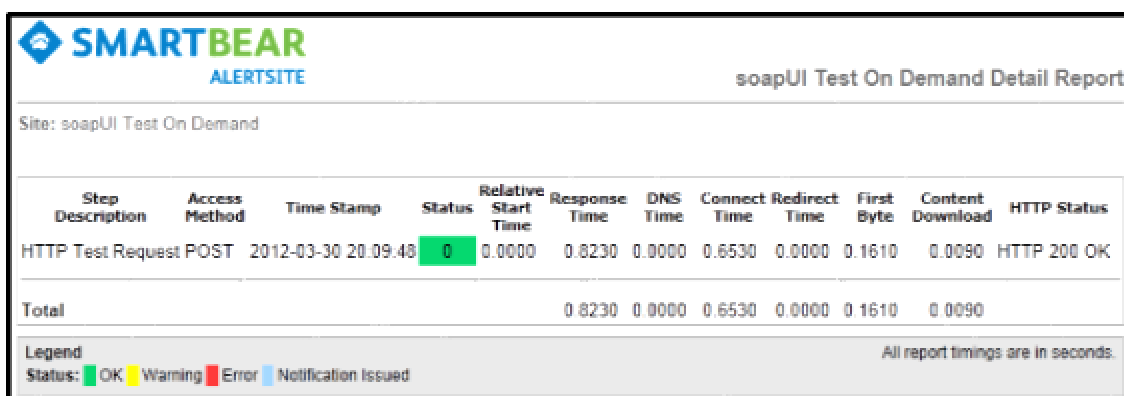
time	type	step	message

0 entries

LoadTest Log LoadTest Assertions Setup Script TearDown Script

Figura 72*. Test de carga del SCR. Caso real*

Finalmente, hay que destacar que el test de carga (se visualiza un ejemplo del test en la *figura 72*) que permite guardar los datos de la prueba de rendimiento. Datos que indicarán al tester si el servicio es capaz de crear muchos registros de forma constante, además de hacerlo en un tiempo aceptable tanto en tablas vacías como en tablas que tienen miles de registros.


SMARTBEAR
ALERTSITE
soapUI Test On Demand Detail Report

Site: soapUI Test On Demand

Step Description	Access Method	Time Stamp	Status	Relative Start Time	Response Time	DNS Time	Connect Time	Redirect Time	First Byte	Content Download	HTTP Status
HTTP Test Request	POST	2012-03-30 20:09:48	0	0.0000	0.8230	0.0000	0.6530	0.0000	0.1610	0.0090	HTTP 200 OK
Total					0.8230	0.0000	0.6530	0.0000	0.1610	0.0090	

Legend
Status: OK Warning Error Notification Issued
All report timings are in seconds.

Figura 73. *Resultados del SCR del caso real*

Si todo ha ido correctamente (véase un ejemplo en la *figura 73*), el tester dará "OK" al proceso del *testing* y habrá terminado después de haber entregado al cliente todos los activos y todos los resultados del *testing*, pudiendo dar por terminado su tarea y avalar que es un servicio de calidad y listo para usarse.

Hay que tener en cuenta que el tester avala la calidad de cada software que realiza, por lo tanto, un tester es tan bueno como su reputación y su experiencia en el proceso de testear el software (algo muy codiciado e importante para muchas empresas y para este sector).

Finalmente, después de todo el proceso del *testing* que se ha realizado sobre el servicio, se puede confirmar: que es un servicio plenamente funcional, tiene un buen nivel de seguridad en todos sus parámetros y sus tiempos de respuesta son correctos. Por lo que este servicio estaría listo para la salida al mercado y funcionar de manera óptima una vez implantado en el sistema.

6. Optimización del proceso del Testing

El proceso del *testing* sufre cambios de forma continuada. Esto se debe a que las empresas siempre buscan formas de optimizar el proceso y que éste sea lo más eficaz posible (tanto para reducir costes como para garantizar un testeo de calidad).

A continuación, explicaré las dos técnicas más extendidas que usan las empresas para actualizar el *testing* y conseguir optimizar el proceso. Junto con cada técnica se explicarán sus ventajas y desventajas, así como en qué consisten.

- La primera técnica son los **estándares**. Consisten en plantillas y documentos que sirven de guía para realizar los activos del tester y los documentos que ha de entregar el cliente (DTAN y DTD), indicando cada sección y los datos que debe tener cada una de estas secciones. Esta técnica ofrece grandes facilidades al tester como:

- Si el tester posee una plantilla para realizar cada uno de los activos, se asegura de que todos los apartados y los datos que requiere el servicio están detallados y no se le pasa nada.

- El tester sabe dónde se encuentra los datos que necesita (los estándares siguen un orden inalterable). Además, estos estándares también suelen tener un formato muy concreto para poner los datos, esto ayuda a evitar las dudas y las confusiones entre cliente y tester.

  - Por ejemplo, cuando se concreta el rango de valores que puede tener un parámetro de tipo "Number" o el formato que ha de tener un parámetro de tipo fecha.

- Es más fácil detectar los fallos de un activo cuando está basado en una plantilla con un orden familiar para el tester.

- Se podrá sacar más potencial de la plantilla si se hace uso de herramientas para realizar algunos activos. Es decir, si el tester sabe dónde se va a encontrar cada dato de un activo porque va a seguir una plantilla, se puede crear scripts que automáticamente los extraigan y creen otros activos únicamente con los datos extraídos que el tester necesite en ese momento.

- El problema de esta técnica es que también tiene importantes desventajas para el tester.

  - El tester también puede tener estándares para hacer los activos y esto hace que el proceso de formación para testear los servicios y el tiempo de la realización de los activos aumente.

  - El tester puede verse muy restringido por la plantilla y alguna peculiaridad que tenga el servicio tenga que ponerla en un documento aparte por no haberse contemplado en el estándar del activo (peculiaridades que pueden cambiar enormemente el proceso de *testing* de un servicio. Como que el servicio tenga varios caminos principales o pueda recibir como entrada tanto datos como ficheros).

- La segunda técnica más empleada por las empresas es el uso del "**Sanity**". Esta técnica consiste en realizar la fase funcional del *testing* únicamente de un caso (suele ser un caso que no acabe en error y pruebe la funcionalidad más importante y básica del servicio). Si este caso funciona con éxito, se realizará el *testing* completo del servicio, pero si el servicio no ha respondido como debería, el tester no continuará hasta que el cliente le haya proporcionado un servicio mínimamente funcional.

- Esta técnica ahorra mucho tiempo al tester ya que, en lugar de realizar los activos funcionales (PLP, ETF, DAT, WSDL y SCF) completos, solo se realizarán con un caso y se ejecutará. De esta forma, si el servicio no responde con la salida esperada del caso(Sanity), el tester no tendrá que rehacer todos los activos cuando el cliente le vuelva a proporcionar el servicio con los errores corregidos.

- Finalmente, desde el punto de vista del cliente, detectar un fallo grave del servicio con esta técnica consigue que el cliente ahorre dinero (dado que con esta técnica ha ahorrado tiempo y esfuerzo al tester al no realizar los activos completos para descubrir el error) y tiempo (al detectar el fallo más pronto, el cliente puede corregirlo antes y el *testing* pueda terminar más pronto de lo que hubiera terminado si no se hubiera aplicado el Sanity).

- La principal desventaja de esta técnica, es que obliga al tester a dejar los activos incompletos y avanzar de un activo a otro rápidamente para dar el visto bueno al Sanity o devolvérselo al cliente para su corrección.

7. Conclusiones

Este trabajo ha explicado el proceso del testing que se ha de llevar según el estándar de factoría, para ello: se han generado documentos, se han detallado los pasos y se han ejecutado las pruebas necesarias en cada una de las fases del proceso del testing. Por lo tanto, se han cumplido con los objetivos inicialmente establecidos y el trabajo ha cumplido con su finalidad.

Al principio, el proceso del *testing* me pareció muy sistemático y mecánico debido al estándar de factoría que usan las consultorías, pero según avanzas se puede observar todos los problemas y complejidades que puede acarrear cada paso y cada fase del *testing* (peculiaridades que varían según el framework).

Es importante aprender lo máximo posible de las herramientas que se van usando durante el proceso, especialmente cuando es un servicio online (ya que usan herramientas más complejas y con múltiples opciones que pueden mejorar el *testing*). En el caso de un servicio local, me parece más complicado el *testing* porque tienes que saber con precisión cómo funcionan los activos para encontrar el fallo según lo que te indica la terminal (por mucho que el tester personalice las herramientas de servicios locales con sus propios activos, el uso de la consola y sus interfaces simples y poco amigables complican la ejecución y detección de errores).

Hay que mencionar que, con respecto a los servicios online, el *soapUI* es muy completo y, aunque requiere de cierto tiempo y practica dominarlo, una vez comprendes todo su potencial, mejora en rapidez y eficacia el *testing*. Con respecto a los servicios locales, requieren de menos activos y los servicios son más simples pero el tester debe saber cómo funciona el servicio de una forma más profunda para poder indicar al cliente el por qué ha fallado (si es que falla) según la salida de la consola, sobre todo si es un servicio BATCH que posee apenas activos y el tester ha de saberlo según lo que ha hecho el desarrollador del servicio.

Para finalizar, tuve que aprender a seguir un proceso concreto (con unos pasos y un orden que se han de seguir al pie de la letra para conseguir un *testing* de calidad) y unos estándares (explicados en el apartado 6) que te imponían para que, tanto el cliente como otros testers sepan qué se ha hecho y qué resultados se han obtenidos de la forma más rápida y clara posible.

8. Bibliografía

1. *Universitat Politécnica de Valencia. Trabajo fin de grado*

   < https://www.upv.es/entidades/ETSINF/info/934549normalc.html>

   [Consulta: 28 de junio de 2017]

2. *Portal oficial de SoapUI*.

   <https://www.soapui.org/>

   [Consulta: 28 de junio de 2017]

3. *Microsoft. Comandos Telnet*

   <https://technet.microsoft.com/en-us/library/bb491013.aspx>

   [Consulta:28 de junio de 2017]

4. *Portal oficial Apache Groovy*

   <http://groovy-lang.org/index.html>

   [Consulta: 28 de junio de 2017]

5. *Portal oficial MySQL*

   <https://www.mysql.com/>

   [Consulta: 28 de junio de 2017]

6. *Thetestingworld. SoapUI*

   <http://www.thetestingworld.com/>

   [Consulta: 28 de junio de 2017]

7. *Microsoft Office*

   <https://products.office.com/es-ES/>

   [Consulta: 28 de junio de 2017]

8. *Portal oficial de excelinfo. Excel y sentencias SQL*

   <http://www.exceleinfo.com/ejecutar-consulta-sql-desde-excel/>

   [Consulta: 28 de junio de 2017]

9. *Oracle. SQL Developer*

<http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html>

[Consulta: 28 de junio de 2017]

10. *W3school. WSDL*

<https://www.w3schools.com/xml/xml_wsdl.asp>

[Consulta: 28 de junio de 2017]

11. *Portal oficial de SoapUI. HermesJMS*

<https://www.soapui.org/jms/getting-started.html>

[Consulta: 28 de junio de 2017]

12. *Portal oficial de la herramienta Putty*

<http://www.putty.org/>

[Consulta: 28 de junio de 2017]

13. *Portal oficial de Generatedata. Creación de valores para base de datos*

<http://www.generatedata.com/ >

[Consulta: 28 de junio de 2017]

14. *Portal oficial de Apache Jmeter*

<http://jmeter.apache.org/>

[Consulta: 28 de junio de 2017]

15. *Portal oficial de la herramienta MobaXterm*

<http://mobaxterm.mobatek.net/>

[Consulta: 28 de junio de 2017]

16. *Portal oficial de la herramienta Smartty*

<http://smartty.sysprogs.com/>

[Consulta: 28 de junio de 2017]

17. *Portal oficial de la herramienta Toad*

<https://www.toadworld.com/products/toad-for-sql-server >

[Consulta: 1 de julio de 2017]

18. Universitat Politécnica de Valencia. Portal RiuNet

< https://riunet.upv.es/handle/10251/10994>

[Consulta: 2 de julio de 2017]