



Plataforma Autonomía de Interoperabilidad

Manual_Consumo_Instrumentales_y_Verificación



**GENERALITAT
VALENCIANA**

Conselleria de Hacienda, Economía
y Administración Pública

DIRECCIÓN GENERAL DE TECNOLOGÍAS
DE LA INFORMACIÓN Y LAS COMUNICACIONES

Versión 021
Marzo de 2025



Unió Europea

Fons Europeu de Desenvolupament Regional
Una manera de fer Europa



Unión Europea

Fondo Europeo de Desarrollo Regional
Una manera de hacer Europa

Índice

Índice	2
1 Control del documento	4
1.1 Información general	4
1.2 Histórico de revisiones	4
1.3 Estado del documento	5
1.4 Alcance	5
1.5 Objetivos	5
1.6 Audiencia	6
1.7 Glosario	6
1.8 Referencias	6
2 Introducción	7
2.1 Alcance del Sistema	9
2.1.1 Separación funcional de servicios	9
2.1.2 Roles del Sistema	10
2.2 Información de Contacto	11
3 Guía de para el consumo de servicios publicados en la PAI	12
3.1 Configuración y requisitos	12
3.1.1 Librería Apache CXF	12
3.1.2 SOAPUI	13
3.1.3 Configuración de eclipse	13
3.2 Generación del cliente	15
3.2.1 Generación del cliente con SOAPui	15
3.2.2 Generación del cliente con el plugin de eclipse	18
3.2.3 Generación del cliente comando wsdl2java	20
3.3 Características y módulos	20
3.3.1 Soporte ws-security con Apache CXF	20
3.3.2 Password Handler	21
3.3.3 Interceptores CXF	22
3.3.4 Firmado	22
3.3.5 Parámetros del servicio	23
3.3.6 Despliegues en JENKINS estructura NICA	24
3.3.7 Generación del Id_trazabilidad	25
4 ANEXO I. Función getSerial()	27
5 ANEXO II. Función getCodigoCati()	28
6 ANEXO III. Desenscriptado de respuestas	29
7 Anexo IV. Firma mediante XMLSignature	30

8	Anexo V. Firmas admitidas de los servicios.....	33
9	Anexo VI. Librería PAI-LIB.....	34

1 Control del documento

1.1 Información general

Título	Manual de usuario de consumo de Servicios Instrumentales y de Verificación
Creado por	DGTIC
Revisado por	
Lista de distribución	
Nombre del fichero	CONSTRUCCION-2- Manual_Usuario_Consumo_Instrumentales_y_de_Verificacion-021.docx

1.2 Histórico de revisiones

Versión	Fecha	Autor	Observaciones
001	02/11/2011	DGTIC	Versión inicial
002	13/02/2012	DGTIC	Adaptación a ENI
003	15/02/2012	DGTIC	Revisión documento
004	12/03/2012	DGTIC	Separación documento por tipo de servicio
005	13/03/2012	DGTIC	Revisión, adaptación plantilla eSirca, URLs
006	05/06/2012	DGTIC	Corrección Username/Token
007	06/07/12	DGTIC	Unificación de Manuales de Verificación e instrumentales
008	14/11/2014	DGTIC	Correcciones en la autoria del documento y actualizado el nombre de la PAI Actualización general del tutorial de generación proyecto, y se actualizan las versiones del software utilizado por unas más recientes.
009	27/01/2015	DGTIC	Agregada instrucción para desactivar el parámetro mustUnderstand en la configuración de firma.
010	11/02/2015	DGTIC	Agregada Instrucción para desactivar la inclusión del nodo InclusiveNamespaces en la firma de la petición.
011	22/04/2016	DGTIC	Se agrega 3.1.1.11 Utilizar un WSDL importado dentro del proyecto local
012	10/03/2017	DGTIC	Se actualiza el manual con CXF3, añadido punto referente al Id_trazabilidad
013	28/04/2017	DGTIC	Adaptación de los puntos de información general a las funcionalidades de los nuevos buses

014	15/09/2017	DGTIC	Actualización generación id_trazabilidad y adición de anexos I y II
015	26/02/2018	DGTIC	Actualización de composición de id_trazabilidad, inclusión de anexo de descriptado y añadidas referencias a la documentación de la Oficina Java
016	26/11/2019	DGTIC	Actualización versión CXF 3.3.4 y jdk 1.8.0_162.
017	28/12/2020	DGTIC	Actualización del descriptado para incluir la inclusión del truststore.
018	23/08/2021	DGTIC	Eliminación referencias a e-Sirca.
019	16/03/2022	DGTIC	Añadido Anexo con la firma XMLSignature
020	13/11/2024	DGTIC	Actualización cuenta de contacto y logos.
021	28/03/2025	DGTIC	Añadido nuevo anexo con la nueva librería PAI-LIB y añadido anexo con las firmas admitidas a validar

1.3 Estado del documento

Responsable aprobación	Fecha

1.4 Alcance

Este documento pretende ser una guía de usuario para el consumo de servicios publicados en la Plataforma Autónoma de Interoperabilidad de la Generalitat Valenciana (a partir de ahora PAI).

1.5 Objetivos

Los objetivos del presente documento son:

- Describir las particularidades y condicionantes que tiene el consumo de los servicios desplegados en el bus de la PAI.
- Describir el ciclo de consumo de servicios publicados en la plataforma, PAI.
- Describir la librería Apache CXF para la generación de clientes de servicio.
- Describir las herramientas necesarias para la generación de un cliente CXF de un servicio publicado en la plataforma.
- Describir como generar un cliente CXF de servicio mediante la herramienta SoapUI.
- Describir como generar un cliente CXF de servicio mediante el plugin de eclipse.

- Dar los conocimientos necesarios para usar el soporte WS-Security de la librería Apache CXF y aplicarlos a los clientes CXF generados.
- Describir brevemente el uso de KeytoolUI para la gestión de keystores y certificados.

1.6 Audiencia

Nombre y Apellidos	Rol
*	CONSUMIDORES DE SERVICIOS

Tabla 1: Audiencia

1.7 Glosario

Término	Definición
PAI	Plataforma Autónoma de Interoperabilidad de la Generalitat Valenciana

Tabla 2: Glosario

1.8 Referencias

Referencia	Título

Tabla 3: Referencias

2 Introducción

La Plataforma Autónoma de Interoperabilidad de la Generalitat Valenciana (PAI) está basada en la herramienta Oracle Service Bus. Las funcionalidades que contempla son:

- Garantizar la autenticidad, confidencialidad, integridad, disponibilidad y trazabilidad de todas las peticiones:
 - **Autenticidad:** Se asegurará la identidad de todos los agentes que intervengan en el proceso de intercambio de datos, de forma que todos ellos estén correctamente identificados en cada intercambio.

Los mecanismos utilizados para identificar a la aplicación demandante del servicio son:

- Identificación mediante certificado digital:
 - Firmar los mensajes intercambiados en cabecera SOAP conforme a estándares WS-Security, permitiendo garantizar la identidad de la aplicación demandante del servicio.
 - Autenticar, mediante certificado cliente en el protocolo SSL, en la invocación al servicio.
- Identificación mediante IP del servidor que realiza la invocación al servicio.
- Identificación mediante api-key para los servicios REST
- **Autorización:** En todas las peticiones, la PAI comprueba que la aplicación consumidora tiene permisos para invocar al servicio.
- **Confidencialidad:**
 - Los mensajes podrán estar cifrados en su totalidad o sólo estar cifrados los bloques que contienen datos sensibles.
 - La información intercambiada está protegida mediante protocolo https y asegurando que no se almacena información personal de ningún ciudadano en la PAI.
- **Integridad:**
 - Validación de firma de las peticiones: Se verifica que la firma del mensaje SOAP es correcta y que dicho mensaje está firmado con el certificado que se incluye en el mensaje.
 - Firma de respuestas: En algunos servicios instrumentales, la PAI firma el mensaje de respuesta con el certificado que indica el proveedor del servicio, para garantizar la validez, la integridad y autenticidad del mensaje de respuesta.
- **Disponibilidad:** La PAI tiene una disponibilidad de 24x7. Es importante recordar que la disponibilidad de cada uno de los servicios, depende de la disponibilidad que tenga el servicio final.
- **Trazabilidad:**
 - La PAI guarda la información necesaria para verificar el intercambio de mensajes entre consumidor y proveedor, pero en ningún caso, almacena información sobre el contenido del intercambio.

- Ante una posible auditoría, la información aportada por la PAI se deberá completar con aquella que permita la recuperación de los datos específicos intercambiados, que deberá conservar tanto el proveedor como el consumidor del servicio.
- Transformación de mensajes. Una de las características que presenta el bus es la adecuación y transformación de los mensajes al formato y tipo esperado por los distintos servicios.
- Orquestación de servicios. La PAI permite definir un flujo diagramado que pueda establecer condiciones de invocación, procesamiento de la información, gestión unificada de errores, almacenamientos de determinados datos de cada petición/respuesta que permitirán tanto la trazabilidad del intercambio, como su explotación posterior para usos estadísticos.
- Herramienta interna de gestión y administración de servicios llamada Gestor de la Plataforma de Interoperabilidad (GePI):

La gestión y control de acceso a los servicios publicados en la PAI se realiza por parte de la DGTIC utilizando GePI, que permite otorgar los permisos adecuados a los consumidores que lo soliciten siguiendo el ciclo de consumo del apartado 2.1.3.

2.1 Alcance del Sistema

2.1.1 Separación funcional de servicios

La separación funcional de los servicios a publicar en la PAI ha permitido identificar dos grupos principales de servicios cuyas características presentan diferencias sensibles y que afectan al modo en que se consumen dichos servicios:

- **Servicios de Verificación:** Estos servicios hacen referencia a las capacidades de verificación que la Administración tiene para evitar solicitar al ciudadano documentación que pueda obrar en su poder en el proceso de interacción entre ambos.
- **Servicios Instrumentales.** Estos servicios hacen referencia a servicios que permiten realizar procesos instrumentales a las distintas entidades de la Administración Pública en su gestión con el ciudadano, como puedan ser notificaciones, pagos, etc.

Tipos de consumidores:

Atendiendo al tipo de consumidor de los servicios publicados en la PAI podemos considerar las siguientes agrupaciones:

- **Organismos solicitantes externos:** Organismos no pertenecientes a la GVA como la Administración General del Estado, Entidades Locales, Comunidades Autónomas, empresas,... Estos entes consumirán los servicios publicados en la plataforma firmando las peticiones a la misma con un certificado en vigor autorizado por la GVA mediante WS-SECURITY.
- **Organismos solicitantes internos:** En este grupo se engloban los consumidores integrados en la Red corporativa de la GVA, distinguiéndose a su vez dos subgrupos:
 - **Organismos GVA:** Consellerias, Organismos dependientes,... que tienen aplicaciones que acceden a los servicios de la PAI, consumirán los servicios firmando las peticiones con un certificado válido autorizado por la GVA, mediante el protocolo WS-SECURITY.
 - **Aplicaciones internas:** Aplicaciones y módulos desarrollados por la DGTIC e instalados en el CA90, consumirán los servicios usando acceso mediante IP o Api-key según requiera el servicio.

2.1.2 Roles del Sistema

Se han definido los siguientes roles:

Rol	Descripción
Consumidor de servicios	Miembro de un organismo o entidad responsable de comunicar a la DGTIC el consumo de determinados servicios.
Desarrollador servicio proveedor	Miembro de un organismo o entidad responsable de desarrollar servicios a publicar a través de la PAI.
Gestor Interoperabilidad	Miembro de la DGTIC responsable de asignar permisos de aplicaciones sobre servicios publicados en la PAI.
Administración técnica PAI	Encargados de especificar políticas y configuraciones generales del bus.

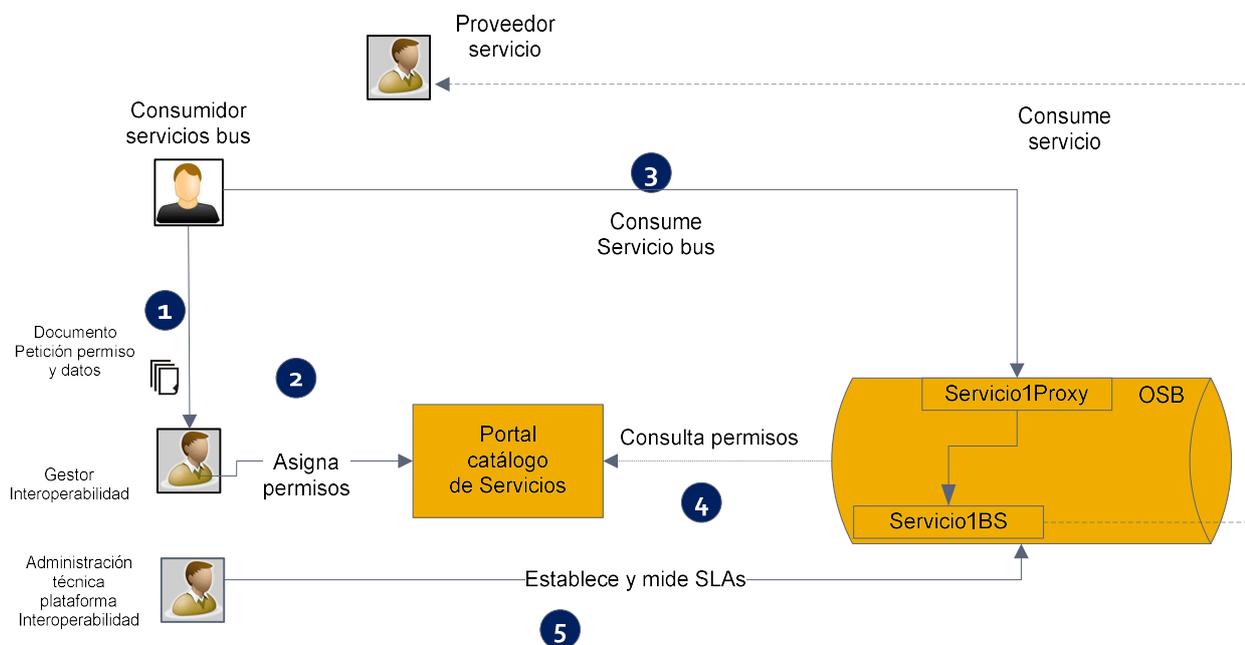
El rol que aplica al presente documento es el de consumidor de servicios de la PAI.

Para el consumo de servicios deberá cumplimentar el formulario de solicitud de servicio donde se indicará, el número de serie del certificado o la IP de los servidores desde los que se realizarán las invocaciones, dependiendo de los requisitos que se recogen en el contrato de integración de cada uno de los servicios.

El Gestor de Interoperabilidad introducirá este dato en la herramienta GePI autorizando de este modo el consumo de un servicio a dicho certificado o IP.

Una vez autorizado el consumo del servicio se podrán realizar peticiones al mismo, todas las peticiones deberán seguir las normativas de seguridad especificadas en el contrato de integración del servicio

El ciclo de consumo completo es el siguiente:



2.2 Información de Contacto

Para cualquier duda o sugerencia relativa a temas de interoperabilidad y el consumo de servicios de la PAI póngase en contacto en la dirección de correo siguiente: esirca_interoperabilidad@gva.es

3 Guía de para el consumo de servicios publicados en la PAI

3.1 Configuración y requisitos

El código fuente ejemplo que se comenta en esta sección se puede bajar desde el portal de la PAI - ¿Cómo usar la plataforma?, el contenido es el siguiente:

- Java jdk 1.8.0_162
- Binarios de la librería Apache CXF, esta librería contiene las clases necesarias para la generación del cliente del servicio y el soporte para securizar las peticiones mediante WS-SECURITY. Usaremos la versión 3.3.4.
 - <http://cxf.apache.org/> ficheros y documentación
- Eclipse: IDE para la implementación del cliente. Usaremos la versión NEON.
 - <http://www.eclipse.org/>
- Plugin de eclipse para Apache CXF, utilizaremos la última versión disponible del mismo.
 - Help -> Install New Software... en eclipse
- SoapUI : Herramienta para testeo de WebServices, usaremos la versión 5.3.0
 - <http://www.soapui.org/>
- Certificado válido emitido por la GVA, en caso de requerir peticiones firmadas, este certificado deberá estar autorizado en el gestor GePI como requisito previo a la utilización del cliente.
- Confirmación del alta para el consumo al servicio tras presentar el Formulario de alta a dicho servicio.
- Configuración de las herramientas

NOTA: Si la aplicación cliente sigue el estándar de la oficina java, es de interés consultar la documentación expuesta en su espacio de confluence, en concreto la relativa a la capa de servicios (190-analisis-capa-servicios-soap).

3.1.1 Librería Apache CXF

Nos descargamos la librería Apache CXF, versión 3.3.4 de la siguiente página web <http://cxf.apache.org/>, una vez descargada, descomprimos los ficheros a una ruta conocida, esta ruta la usaremos para configurar el plugin de eclipse CXF y el soapUI.

La carpeta bin contiene los binarios que usa la librería para la generación de los clientes a partir del wsdl del servicio, etc., debemos agregar la ruta de esta carpeta a la variable PATH

Linux:

```
export CXF_HOME=[ruta_carpeta_cxf]
export PATH=$CXF_HOME/bin:$PATH
```

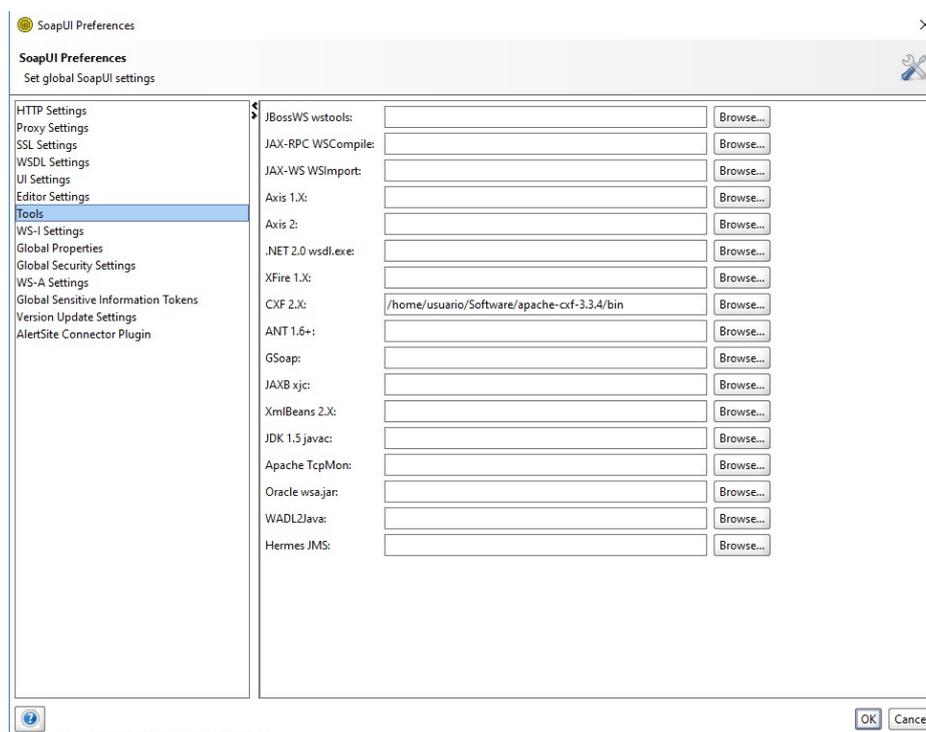
Windows

```
set CXF_HOME=[ruta_carpeta_cxf]
set PATH=%CXF_HOME%;%PATH%
```

3.1.2 SOAPUI

SoapUI es una herramienta para el testeo y generación de Servicios Web, soporta múltiples librerías y especificaciones WS-*, para poder generar el cliente CXF deberemos indicarle a la herramienta la ruta donde residen la librería que previamente hemos descargado, para ello:

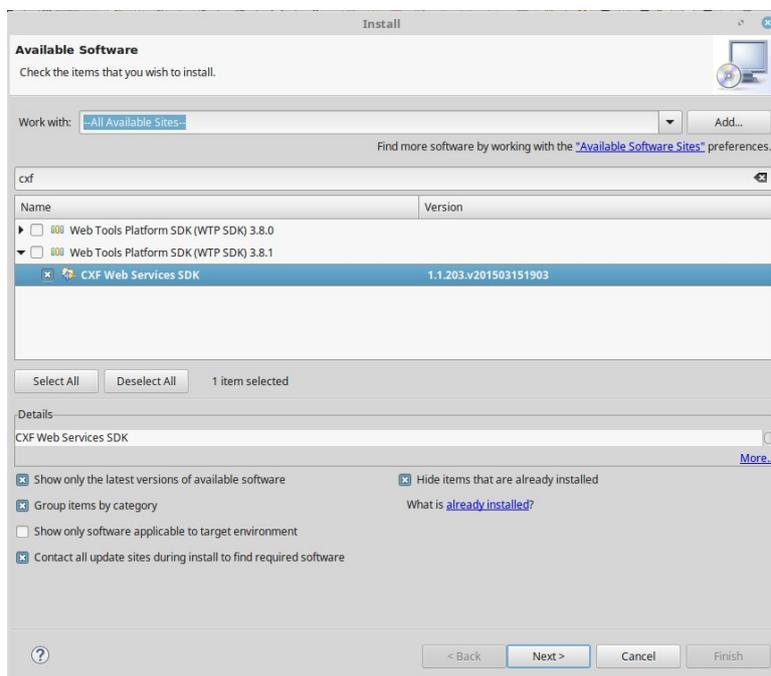
- Descargamos e instalamos el SoapUI 5.3.0 de la web indicada.
- Arrancamos la aplicación y vamos a File -> Preferences -> Pestaña tools.
- Apuntamos CXF a donde hayamos descomprimido los binarios de CXF.



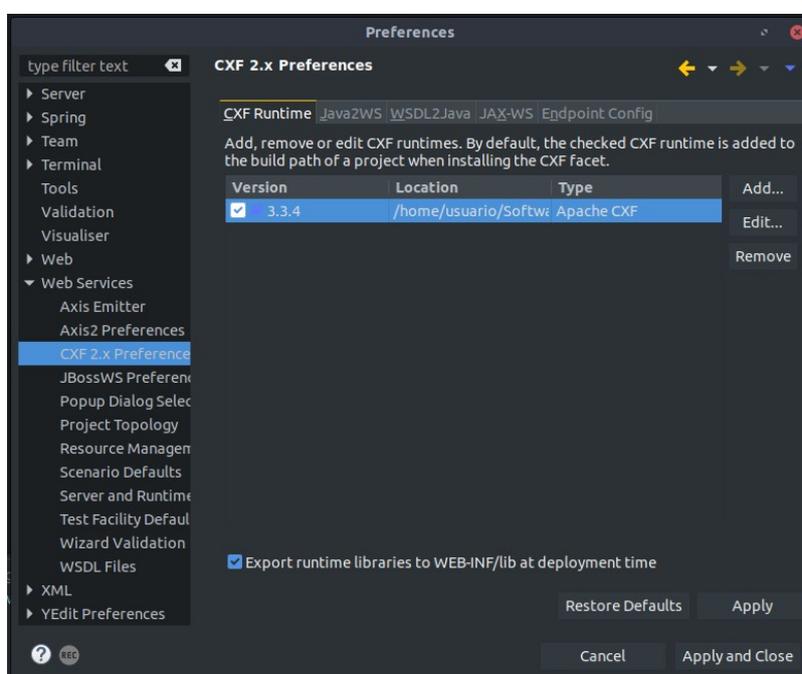
3.1.3 Configuración de eclipse

Descargamos eclipse NEON, preferentemente la versión “Eclipse IDE for Java EE Developers”, si no lo tenemos ya instalado, (en caso de una versión anterior el plugin debería soportarlo también), El plugin de CXF ya viene instalado por defecto en la versión indicada, pero en caso de que estemos trabajando en una versión que no tenga el plugin se puede instalar siguiendo los pasos detallados a continuación:

- Accedemos al menú Help -> Install new software ...
- Desplegamos la pestaña work with y seleccionamos todos los sitios disponibles.
- Esperamos a que cargue los datos.
- Buscamos CXF e instalamos la última versión del plugin.
- Reiniciamos eclipse cuando lo indique el IDE.



- Vamos al menú Window -> Preferences.
- Desplegamos nodo Web Services.
- Indicamos donde se encuentra la librería CXF que previamente habíamos descargado.



3.2 Generación del cliente

Entre las distintas soluciones posibles para la generación de clientes de servicios web mediante CXF, en este documento vamos a tratar la generación desde SoapUI y desde Eclipse. No obstante, es recomendable revisar la documentación de la Oficina Java donde se trata la generación de clientes mediante Maven a través del plugin de CXF (<http://confluence.gva.es/display/OJ/Documentos>), concretamente en su documento de análisis de la capa de integración.

3.2.1 Generación del cliente con SOAPui

Vamos a proceder a la generación del cliente del servicio, como servicio de pruebas vamos a usar el servicio de envío de correos mediante SMTP, para ello necesitaremos la url del descriptor del servicio en adelante WSDL, el WSDL describe los métodos del servicio y los parámetros que recibe y envía, así como las excepciones que produce el servicio.

La url del WSLD del servicio de SMTP es la siguiente:

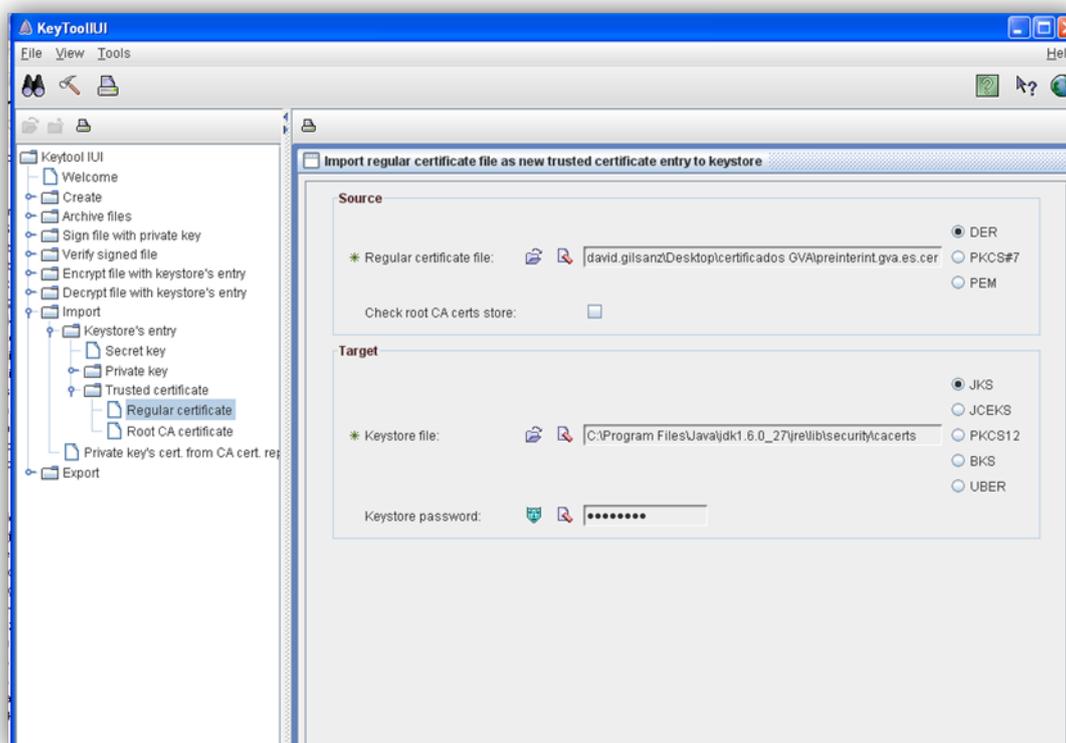
https://instrumental-pre.gva.es/pai_bus_ins/GVA/SMTPService_v2_00?wsdl

Como se puede observar el acceso al mismo se hace mediante HTTPS por lo que deberemos tener la **ruta de certificación** del certificado de servidor en el keystore de certificados de confianza.

Para ello, usaremos una herramienta de gestión de certificados y keystores llamada keytool-ui accesible en la siguiente url <http://code.google.com/p/keytool-iii/> y la arrancamos.

Seguimos los siguientes pasos:

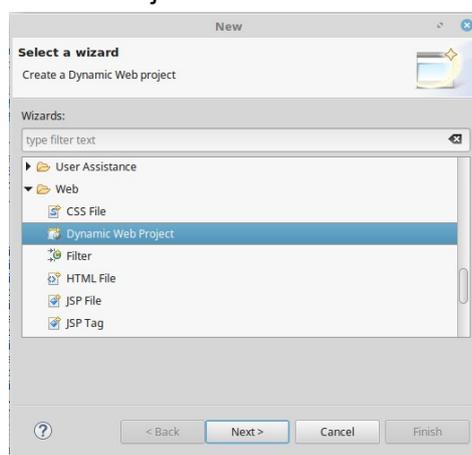
- Descargamos certificado de preproducción mediante un navegador, (candado de la barra de direcciones), y lo añadimos a \$JAVA_HOME\jre\bin\lib\security\cacerts usando keytool mediante la opción import -> trusted certificate -> Regular certificate como se muestra en la imagen. En ocasiones hay que cambiar temporalmente el nombre del fichero cacerts, (y ponerle la extensión JKS), para que lo pueda abrir la herramienta keytool-ui (existen otras herramientas como portecle o el mismo keytool management que es posible ejecutar como comando de consola). Una vez modificado, hay que quitarle la extensión, y dejarlo con el nombre original.
- El password por defecto del keystore cacerts es ... *'changeit'*



NOTA: Es recomendable conocer Java instalado en la máquina e intentar realizar la ejecución del software que generará el cliente con el Java correcto para su desarrollo posterior.

Ahora el wsdl podrá ser accesible desde la aplicación cliente, por lo que procedemos a generar el cliente, para ello hacemos lo siguiente:

- Creamos un nuevo proyecto con eclipse
 - File -> New -> Other...
 - Web -> Dinamic Web Project.

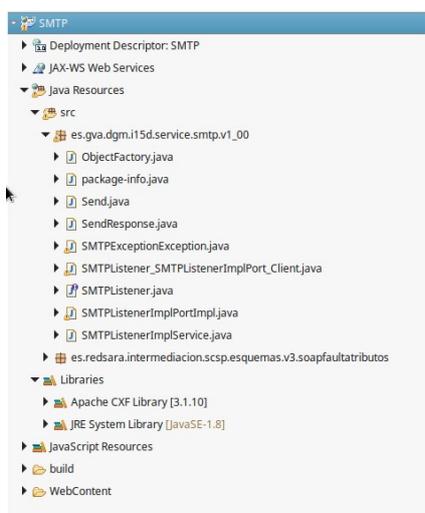


- Generamos el cliente con el SoapUI
 - Vamos al menú Tools -> Apache CXF.
 - Introducimos la url del wsdl del servicio.
 - Introducimos un nombre de paquete según Otp : es.gva.dgm.ws.xxxxxx
 - Apuntamos a la carpeta src del proyecto de eclipse recién creado.
 - Click en solo cliente y pulsamos generate.

Refrescamos la carpeta src del proyecto de eclipse y ya tenemos el cliente del servicio generado. La única clase que tocaremos será xxxxxxxxxxxx_Client.java, que es la que invoca al método elegido proporcionándole los parámetros requeridos.

Básicamente lo que ha ocurrido es que el SoapUI, a partir de la descripción de servicio y usando las herramientas de la librería CXF, nos ha generado un conjunto de clases que nos permiten consumir el servicio de SMTP.

Este cliente generado no tiene seguridad de ningún tipo añadida, si el servicio final no tuviese políticas de seguridad activadas podríamos consumir el servicio con este cliente directamente.



3.2.1.1 Dependencias

Como podemos observar el IDE muestra errores de compilación, eso es debido a que no hemos añadido al proyecto las dependencias de la librería CXF, para ello haremos lo siguiente:

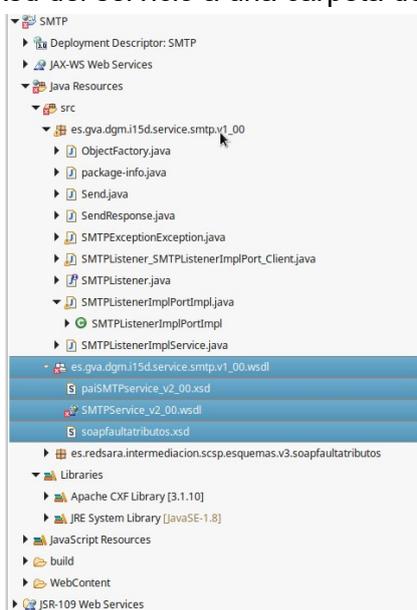
- Creamos carpeta lib colgando de la raíz del proyecto
- Añadimos los jars de la librería CXF:
 - Los jars se encuentran en la carpeta lib de la librería CXF.
 - En este enlace, <http://cxf.apache.org/docs/a-simple-jax-ws-service.html> se muestran los jars necesarios en función de la configuración del proyecto.

- Los mínimos para el cliente indicados en la imagen siguiente.
- Añadimos al classpath los jars añadidos a la carpeta lib pulsando botón derecho sobre el proyecto y pulsamos en Java Build Path, navegamos hasta la carpeta lib y añadimos las librerías.

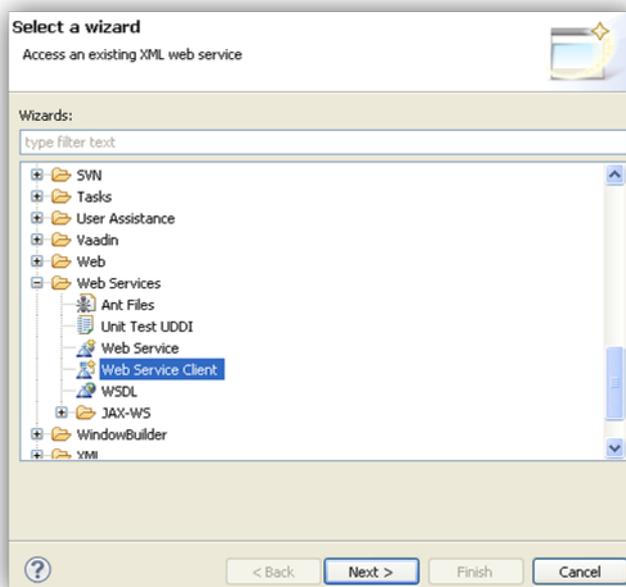
NOTA: es importante que configuremos en este apartado el JRE que hemos configurado con el certificado de preproducción y no otro, de lo contrario no funcionará correctamente el cliente.

3.2.2 Generación del cliente con el plugin de eclipse

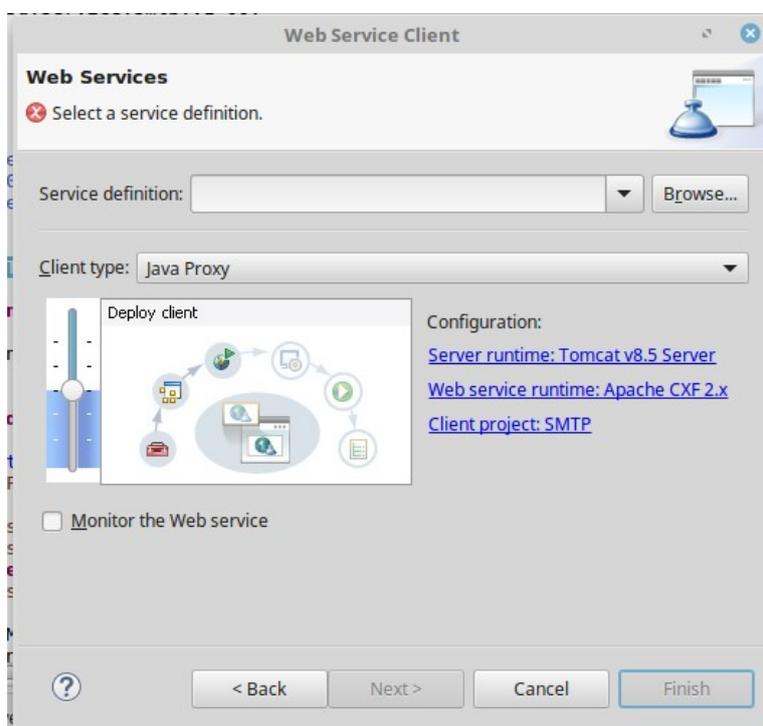
Nos descargamos el wsdl y el xsd del servicio a una carpeta del proyecto



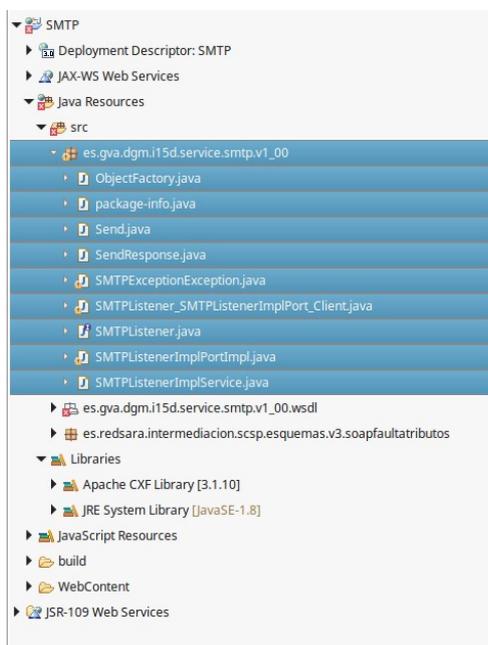
- Sobre el wsdl botón derecho
- New -> Other ...
- Web Services -> Web Service client



- Seleccionamos CXF como web service runtime
- Generamos el cliente



- Copiamos los interceptores y los parámetros de los puntos anteriores en función del tipo de autenticación que queremos usar.



- Ahora podemos ejecutar el cliente del servicio haciendo click con el botón derecho del ratón sobre el proyecto y elegimos una de las dos opciones.
 - Run as -> java application
 - Debug as -> java application

Para consumir el resto de servicios publicados en la plataforma seguiremos los mismos pasos que se han descrito en el presente manual, únicamente cambiarían los wsdl de servicio y los parámetros de los métodos a configurar según la documentación de cada servicio.

3.2.3 Generación del cliente comando *wsdl2java*

Para la creación del cliente utilizando la consola, debemos usar el comando “wsdl2java” disponible en CXF y configurar los parámetros de creación que determinemos.

```
./wsdl2java -client https://instrumental-pre.gva.es/pai_bus_ins/GVA/SMTPService_v2_00?wsdl
```

Esto creará los fuentes necesarios para la integración con el servicio, incluyendo una clase con un método main que contiene una invocación por defecto al servicio.

3.3 Características y módulos

Se tratan a continuación las diferentes características y su aproximación en CXF para completar la implementación del cliente siguiendo

3.3.1 Soporte *ws-security* con Apache CXF

Apache CXF da soporte a WS-SECURITY mediante interceptores de entrada y salida en las invocaciones a servicios. En el interceptor de salida, (ya que estamos consumiendo el

servicio), configuramos el tipo de seguridad a utilizar, en caso del uso de certificados para el firmado deberemos configurar el keystore que almacena el certificado que usamos para firmar la petición, para ello haremos lo siguiente:

- Creamos un fichero de texto en la carpeta src del proyecto con nombre `client_sign.properties`, aquí se configura el keystore a utilizar, el alias de la clave privada para firmar, password del keystore, etc. con las siguientes líneas:

```
#Provider en nuestro caso Merlin  
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin  
#Ruta al keystore que almacena el certificado a usar  
org.apache.ws.security.crypto.merlin.file=d:\\somefile.pfx  
#Tipo de keystore habitualmente JKS o PKCS12  
org.apache.ws.security.crypto.merlin.keystore.type=PKCS12  
#Clave del keystore  
org.apache.ws.security.crypto.merlin.keystore.password=secreto
```

Para más información sobre certificados, keystores, etc. Visitar el siguiente enlace:

- <http://cxf.apache.org/docs/ws-security.html> enlace con la documentación, introducción al uso de certificados, generación de los mismos con keytool, configuración del keystore y uso de interceptores CXF.

3.3.2 Password Handler

Apache CXF gestiona el uso de las claves de los certificados mediante una interfaz llamada `CallbackHandler`, en nuestro cliente deberemos proporcionar una clase que implemente dicha interfaz para poder usar la clave privada del certificado a fin de firmar las peticiones, para ello:

- Creamos la clase `WsPasswordHandler` en el paquete del cliente y copiamos el siguiente código sustituyendo “secreto” por la clave real del certificado a usar. En entornos productivos se debe parametrizar obteniéndose dicha clave del fichero de propiedades, pero entiéndase este ejemplo como base de creación, en la que se pretende mostrar los pasos básicos guía.

```
package es.gva.dgm.ws.smtp;  
  
import java.io.IOException;  
import javax.security.auth.callback.Callback;  
import javax.security.auth.callback.CallbackHandler;  
import javax.security.auth.callback.UnsupportedCallbackException;  
import org.apache.ws.security.WSPasswordCallback;  
public class WsPasswordHandler implements CallbackHandler {  
  
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {  
        for(Callback callback: callbacks) {  
            WSPasswordCallback pwdCallback = (WSPasswordCallback)callback;  
            int usage = pwdCallback.getUsage();  
            if(usage == WSPasswordCallback.SIGNATURE || usage == WSPasswordCallback.DECRYPT) {  
                pwdCallback.setPassword("secreto");  
            }  
        }  
    }  
}
```

```
}  
}  
}  
}
```

3.3.3 Interceptores CXF

Existen dos tipos de interceptores CXF.

- De salida (out)
 - Intercepta peticiones a servicios y permite:
 - WSHandlerConstants.USERNAME_TOKEN (usuario/contraseña)
 - WSHandlerConstants.TIMESTAMP (sello)
 - WSHandlerConstants.SIGNATURE (firma)
 - WSHandlerConstants.ENCRYPT (encriptación)
 - ...
- De entrada (in)
 - Proceso a la inversa para las respuestas del servicio.
 - WSHandlerConstants.ENCRYPT (desencriptar)
 - WSHandlerConstants.SIGNATURE (validación firma)
 - ...

WSHandlerConstants.xxxxx contiene las constantes predefinidas en CXF, siendo SIGNATURE para el firmado con certificado.

NOTA: En el Anexo III

3.3.4 Firmado

Para firmar la petición mediante WS-SECURITY añadiremos el siguiente código al cliente:

```
//Obtenemos el endpoint del servicio  
org.apache.cxf.endpoint.Client client = ClientProxy.getClient(port);  
org.apache.cxf.endpoint.Endpoint cxfEndpoint = client.getEndpoint();  
  
//Creamos un HashMap para almacenar las propiedades del interceptor  
Map<String, Object> outProps = new HashMap<String, Object>();  
//Añadimos la acción que vamos a realizar, en este caso firmar  
outProps.put(WSHandlerConstants.ACTION, WSHandlerConstants.SIGNATURE);  
//Alias del certificado a usar como USER (ESPECIFICACION WS_SECURITY)  
outProps.put(WSHandlerConstants.USER, "alias_clave_certificado*");  
//Alias del certificado a usar como SIGNATURE_USER (ESPECIFICACION WS_SECURITY)  
outProps.put(WSHandlerConstants.SIGNATURE_USER, "alias_clave_certificado*");  
//Paquete y clase del PasswordHandler que previamente hemos añadido  
outProps.put(WSHandlerConstants.PW_CALLBACK_CLASS, "es.gva.dgm.ws.smtp.WsPasswordHandler");  
//Fichero de propiedades que almacena los datos del keystore  
outProps.put(WSHandlerConstants.SIG_PROP_FILE, "client_sign.properties");  
//Indicamos que haga referencia por ID a la firma (ESPECIFICACION WS_SECURITY)  
outProps.put(WSHandlerConstants.SIG_KEY_ID, "DirectReference");  
// Con esta propiedad configurada como "false" conseguimos que se genere la firma
```

```
//sin nodos "inclusiveNamespaces")
outProps.put(WSHandlerConstants.IS_BSP_COMPLIANT, "false");
// MUSTUNDERSTAND = 0
outProps.put(WSHandlerConstants.MUST_UNDERSTAND, "0");
//Creamos el interceptor y se lo asociamos al endpoint del cliente
WSS4JOutInterceptor wssOut = new WSS4JOutInterceptor(outProps);
cxfEndpoint.getOutInterceptors().add(wssOut);
```

* El alias del certificado lo podemos consultar mediante keytool-ui visualizando el keystore, o mediante la orden:

```
keytool -list -v -keystore 'somefile.pfx' -storetype pkcs12
```

Siendo 'somefile.pfx' el certificado usado en el ejemplo para firmar la petición.

3.3.5 Parámetros del servicio

En este punto tenemos un cliente operativo con la configuración de seguridad asociada mediante un interceptor cxf de salida, únicamente nos queda proporcionar parámetros válidos a la llamada al servicio para poder ejecutar el cliente, para ello haremos lo siguiente:

- Introducimos los parámetros de la llamada al servicio según documentación funcional del servicio.
- En el caso del servicio SMTP modificaremos el cliente con los siguientes parámetros:

```
//From el que manda el email
java.lang.String _send_sendFrom = "simac@gva.es";
//Subject del email
java.lang.String _send_subject = "prueba cliente smtp";
//Array de destinatarios del correo
java.util.List<java.lang.String> _send_sendTo = new ArrayList<String>();
_send_sendTo.add("foovar@email.com");
//CC destinatarios en copia (opcional)
java.util.List<java.lang.String> _send_sendCC = null;
//BCC destinatarios en copia oculta (opcional)
java.util.List<java.lang.String> _send_sendBCC = null;
//Urls de los adjuntos del correo, deben ser visibles desde el servidor de correo //(opcional)
java.util.List<java.lang.String> _send_attachments = null;
//Texto del correo
java.lang.String _send_texto = "Hola mundo";
```

A continuación, podemos ejecutar el cliente del servicio haciendo click con el botón derecho del ratón sobre el proyecto y elegimos una de las dos opciones.

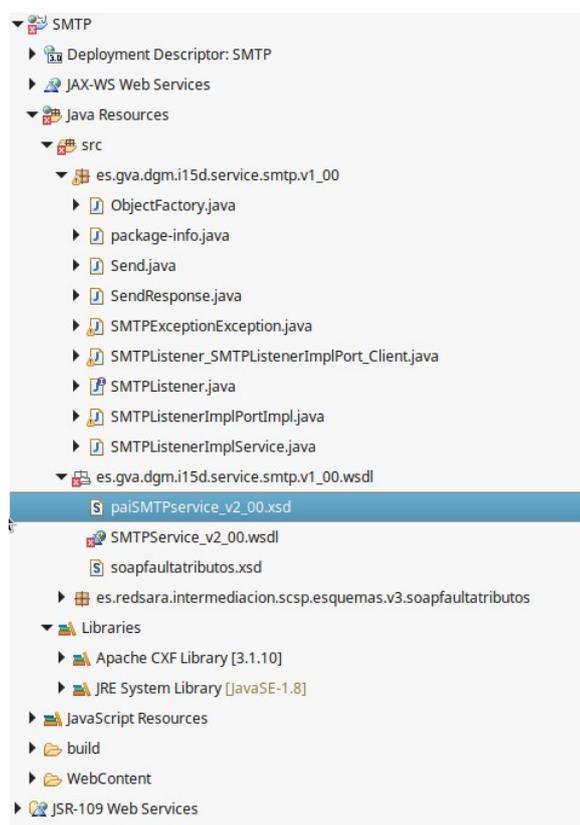
- Run as -> java application
- Debug as -> java application

3.3.6 Despliegues en JENKINS estructura NICA

Puede ser que prefiramos utilizar un WSDL que esté importado dentro del proyecto, en lugar de recuperarlo directamente de la URL del servicio final en tiempo de compilación. En este punto vamos a explicar la manera de configurarlo así.

Primero tenemos que importar el wsdl con sus esquemas en una carpeta que esté dentro del classpath del proyecto. Por ejemplo, esto se puede realizar directamente creando una carpeta con el nombre “wsdl” que esté a la misma altura que el paquete “es.gva.dgm.i15d.service.smtp.v1_00.xxxxxx” que creamos anteriormente, y copiar dentro el WSDL y XSD’s que contenga el servicio.

Una vez copiado al refrescar el proyecto en el eclipse, aparecerá el nuevo paquete como se muestra en la siguiente imagen:



Seguidamente tendremos que indicar en el código del cliente, que el WSDL y XSD’s que debe recuperar están en la ruta donde lo hemos colocado. Para ello debemos modificar en nuestro ejemplo el archivo “SMTPListener_SMTPListenerImplPort_Client.java”.

En el código de este archivo, la primera línea del “main” indica de donde tiene que coger el WSDL de la siguiente manera a través de la variable WSDL_LOCATION:

```
URL wsdlURL = SMTPListenerImplService.WSDL_LOCATION;
```

Nosotros para nuestro ejemplo deberemos modificarlo para que utilice la función `getResource`, que recupera la ruta de nuestro proyecto (para que sea independiente de la máquina donde esté instalado) y además le pasamos como parámetro la ruta dentro del classpath donde lo hemos colocado. Quedaría de la siguiente manera:

```
URL wsdlURL =  
SMTPListener_SMTPLListenerImplPort_Client.class.getResource("/es/gva/dgm/i15d/service/smtp/v1_00/wsdl/  
SMTPService_v2_00.wsdl");
```

Con esto ya es suficiente para que funcione sin cargar el WSDL en tiempo de compilación.

3.3.7 Generación del `Id_trazabilidad`

El consumo de servicios Instrumentales de la PAI requiere la inclusión de la cabecera `Id_trazabilidad`, la composición y valor de esta cabecera depende del Bus (entorno) al que se quiera acceder:

- Innovación: `-[CODIGO_CATI]-[TIMESTAMP]`, por ejemplo:

```
<Id_trazabilidad xmlns="http://dgti.gva.es/interoperabilidad">CATI-20140130184955000</Id_trazabilidad>
```

- Instrumental: `[SERIAL_NUMBER]-[CODIGO_CATI]-[TIMESTAMP]`

```
<Id_trazabilidad xmlns="http://dgti.gva.es/interoperabilidad">68254ed222d65eeb-CATI-20140130184955000</Id_trazabilidad>
```

Siendo las variables:

- `[CODIGO_CATI]`: Valor del Código asociado a la aplicación en CATI
- `[SERIAL_NUMBER]`: Valor del serial del certificado en hexadecimal.
- `[TIMESTAMP]`: Marcha de tiempo en formato `YYYYMMddHHmmssSSS`

En este ejemplo al ser un cliente para el servicio de SMTP Instrumental deberemos incluir en nuestro código las siguientes líneas:

```
java.util.Date date = Calendar.getInstance().getTime();  
SimpleDateFormat dt1 = new SimpleDateFormat("yyyyMMddHHmmssSSS");  
  
//Obtenemos el endpoint del servicio  
org.apache.cxf.endpoint.Client client = ClientProxy.getClient(port);  
org.apache.cxf.endpoint.Endpoint cxfEndpoint = client.getEndpoint();  
//Creamos lista de cabeceras personalizadas  
List<Header> headersList = new ArrayList<Header>();  
Header testSoapHeader1 = new Header(new QName("http://dgti.gva.es/interoperabilidad",  
"Id_trazabilidad"), getSerial() + "-" + getCodigoCati() + "-" + dt1.format(date), new  
JAXBDataBinding(String.class));  
headersList.add(testSoapHeader1);  
client.getRequestContext().put(Header.HEADER_LIST, headersList);
```

La composición de los parámetros del `Id_trazabilidad` se realiza a raíz de unas variables que se han de conocer durante la petición de alta al servicio correspondiente.

NOTA: Hay que tener en cuenta que los parámetros del `Id_trazabilidad` varía dependiendo del bus que se quiera consumir.

El valor del número de serie del certificado NO se debe coger de una variable estática del `.properties`, el valor se debe obtener cada vez al consultar los datos del certificado

almacenado en el keystore, así evitaremos tener que cambiar el .properties y tener que volver a desplegar la aplicación, cuando se renueve el certificado.

Las funciones `getSerial()` y `getCodigoCati()`, recuperarán el número de serie del certificado que se esté utilizando y el código CATI de la aplicación utilizada. La definición de estas funciones se puede ver en los apartados: **ANEXO I. Función `getSerial()`** y **ANEXO II. Función `getCodigoCati()`**.

4 ANEXO I. Función getSerial()

Esta función carga un fichero PROPERTIES para poder recuperar los valores del certificado que le indiquemos. Con estas properties, accede al certificado con el objetivo de poder conseguir su número de serie, el cuál es necesario para formar el id_trazabilidad en las peticiones del bus instrumental.

```
public static String getSerial() throws Exception{
    String serialNumber;

    Properties propertiesSN = new Properties();
    propertiesSN.load(new
FileInputStream("${asa.conf}+[NOMBRE_FICHERO_PROPERTIES]));
    char[] password =
propertiesSN.getProperty([PROPIEDAD_PASSWORD_CERTIFICADO]).toCharArray();

    KeyStore p12 = KeyStore.getInstance("pkcs12");
    p12.load(new FileInputStream([PROPIEDAD_RUTA_CERTIFICADO.p12]),
password);

    Enumeration e = p12.aliases();
    String alias = (String) e.nextElement();

    KeyStore.PrivateKeyEntry keyEntry = (KeyStore.PrivateKeyEntry)
p12.getEntry(alias, new KeyStore.PasswordProtection(password));

    X509Certificate cert = (X509Certificate) keyEntry.getCertificate();

    BigInteger bi = cert.getSerialNumber();
    serialNumber = bi.toString(16);

    return serialNumber;
}
```

5 ANEXO II. Función getCodigoCati()

Esta función carga un fichero PROPERTIES para poder recuperar el valor del CODIGO CATI de la aplicación, el cuál es necesario para formar el id_trazabilidad.

```
public static String getCodigoCati() throws IOException {
    String cati;

    Properties = new Properties();
    properties.load(new
FileInputStream("${asa.conf}+[NOMBRE_FICHERO_PROPERTIES]));

    cati = properties.getProperty([PROPIEDAD_CODIGO_CATI]);

return cati;
```

6 ANEXO III. Desenscriptado de respuestas

Existen ocasiones en las cuales las respuestas vendrán cifradas por contener información sensible, cuando esto ocurra será necesario incluir un interceptor de entrada, siguiendo las siguientes líneas:

```
/* Configurando interceptor de ENTRADA para el encriptado*/
Map<String, Object> inProps = new HashMap<String, Object>();
inProps.put(WSHandlerConstants.ACTION, WSHandlerConstants.SIGNATURE + " " +
WSHandlerConstants.ENCRYPT);
inProps.put(WSHandlerConstants.SIG_PROP_FILE, [URL_Properties]);
inProps.put(WSHandlerConstants.DEC_PROP_FILE, [URL_Properties]);
inProps.put(WSHandlerConstants.ALLOW_RSA15_KEY_TRANSPORT_ALGORITHM, "true");
inProps.put(WSHandlerConstants.PW_CALLBACK_REF, [Password_Handler]);
// Incluyendo propiedades
Properties cxfPropsIn = new Properties();
cxfPropsIn.setProperty("org.apache.ws.security.crypto.provider",
"org.apache.ws.security.components.crypto.Merlin");
cxfPropsIn.setProperty("org.apache.ws.security.crypto.merlin.keystore.type", "jks");
cxfPropsIn.setProperty("org.apache.ws.security.crypto.merlin.keystore.alias",
prop.getProperty("org.apache.ws.security.crypto.merlin.keystore.alias"));
cxfPropsIn.setProperty("org.apache.ws.security.crypto.merlin.keystore.password",
prop.getProperty("org.apache.ws.security.crypto.merlin.keystore.password"));
cxfPropsIn.setProperty("org.apache.ws.security.crypto.merlin.keystore.file",
prop.getProperty("org.apache.ws.security.crypto.merlin.keystore.file"));
cxfPropsIn.setProperty("org.apache.ws.security.crypto.merlin.truststore.file",
prop.getProperty("org.apache.ws.security.crypto.merlin.truststore.file"));
cxfPropsIn.setProperty("org.apache.ws.security.crypto.merlin.truststore.password",
prop.getProperty("org.apache.ws.security.crypto.merlin.truststore.password"));
cxfPropsIn.setProperty("org.apache.ws.security.crypto.merlin.truststore.type",
prop.getProperty("org.apache.ws.security.crypto.merlin.truststore.type"));

final Crypto cryptoIn = CryptoFactory.getInstance(cxfPropsIn);
// Creando WSS4J de ENTRADA
WSS4JInInterceptor wssIn = new WSS4JInInterceptor(inProps) {
    @Override
    protected Crypto loadCryptoFromPropertiesFile(String propFilename, RequestData
reqData)
        throws WSSecurityException {
        return cryptoIn;
    }
};
cxfEndpoint.getInInterceptors().add(wssIn);
```

Se deberá añadir a la condición del WsPasswordHandler que las llamadas encriptadas sean tratadas.

```
if (usage == WSPasswordCallback.SIGNATURE || usage == WSPasswordCallback.DECRYPT)
```

Para realizar el descifrado de la respuesta del mensaje, se han de incluir en el fichero de propiedades que utiliza la aplicación para la configuración de firma.

```
org.apache.ws.security.crypto.merlin.truststore.file=${sys:asa.conf}/certificado/trust_store_PA1
.jks
org.apache.ws.security.crypto.merlin.truststore.password=*****
org.apache.ws.security.crypto.merlin.truststore.type=JKS
```

NOTA para aplicaciones de la DGTIC: Existe un jks instalado por la PAI en los servidores. Para su uso se debe notificar a la PAI (pai@gva.es) y con ello se realizará la autorización y la entrega de los valores concretos para su correcta configuración

7 Anexo IV. Firma mediante XMLSignature

En ocasiones los servicios no serán firmados mediante WS-SECURITY, sino mediante XMLSignature. En estos casos se deberá incluir un interceptor de salida al cliente, siguiendo las siguientes líneas:

```
String result = "";
    String rutaKeystore = properties.getProperty("org.apache.ws.security.crypto.merlin.file");
    String keystorePassword =
properties.getProperty("org.apache.ws.security.crypto.merlin.keystore.password");
    String keyAlias =
properties.getProperty("org.apache.ws.security.crypto.merlin.keystore.alias");
    String aliasPassword =
properties.getProperty("org.apache.ws.security.crypto.merlin.keystore.alias.password");
    String keystoreType =
properties.getProperty("org.apache.ws.security.crypto.merlin.keystore.type");
    TransformUtils transformer = new TransformUtils();
    SignatureUtils sigutil = new SignatureUtils();
    Document doc = null;
    try {
        doc = transformer.string2Document(currentEnvelope);
    } catch (Exception e) {
        LOG.error("Error al generar el documento ",e);
    }
    try {
        LOG.debug("Generando la firma");
        sigutil.firma(doc, rutaKeystore, keystorePassword, keyAlias, aliasPassword, keystoreType);

        LOG.debug("Generando la cadena");
        result = transformer.document2String(doc);
    } catch (Exception e) {
        LOG.error("Error al generar la firma", e);
    }
    return result;
```

Para configurar la firma el interceptor cargará los siguientes datos del fichero de propiedades:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=alias
org.apache.ws.security.crypto.merlin.keystore.alias.password=password
org.apache.ws.security.crypto.merlin.file=archivo.jks
```

Y, por último, el interceptor hará uso de un método **firma** que generará la propia firma para su envío en la petición:

```
public String firma(Document msg, String rutaKeystore, String keystorePassword, String alias,
String aliasPassword, String keystoreType) throws Exception {
```

```
        NodeList                                body                                =
msg.getElementsByTagNameNS("http://schemas.xmlsoap.org/soap/envelope/", "Body");
        NodeList                                header                                =
msg.getElementsByTagNameNS("http://schemas.xmlsoap.org/soap/envelope/", "Header");

        if(header.getLength() == 0){
            //Insertar header al documento
            Element p = msg.createElementNS("http://schemas.xmlsoap.org/soap/envelope/",
"Header");
            p.setPrefix(body.item(0).getPrefix());
            body.item(0).getParentNode().insertBefore(p, body.item(0));
        }

        // Crear un XMLSignatureFactory que se utilizara para generar la firma
XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");

        /* Crear la referencia al elemento que se va a firmar, con las transformaciones
necesarias*/
        ArrayList<Transform> transformList = new ArrayList<Transform>();
        TransformParameterSpec transformSpec = null;
        Transform excl4nTransform = fac.newTransform("http://www.w3.org/2001/10/xml-exc-
c14n#", transformSpec);
        transformList.add(excl4nTransform);

        // se pasa la ID de (idElemento) como parámetro
String idElemento = UUID.randomUUID().toString();
        Reference ref = fac.newReference("#" + idElemento,
fac.newDigestMethod(DigestMethod.SHA1, null), transformList, null, null);
        // Crear el elemento SignedInfo.
        SignedInfo si =
fac.newSignedInfo(fac.newCanonicalizationMethod(CanonicalizationMethod.EXCLUSIVE,
(C14NMethodParameterSpec) null),
fac.newSignatureMethod(SignatureMethod.RSA_SHA1,null),
Collections.singletonList(ref));

        /* Cargar el keystore que con la clave que se utilizarán para realizar la firma.*/
KeyStore ks = KeyStore.getInstance(keystoreType);
ks.load(new FileInputStream(rutaKeystore), keystorePassword.toCharArray());
        KeyStore.PrivateKeyEntry keyEntry = (KeyStore.PrivateKeyEntry) ks.getEntry(alias,
new KeyStore.PasswordProtection(aliasPassword.toCharArray()));
        X509Certificate cert = (X509Certificate) keyEntry.getCertificate();
        // Crear el elemento KeyInfo que contiene X509Data.
        KeyInfoFactory kif = fac.getKeyInfoFactory();
        List<X509Certificate> x509Content = new ArrayList<X509Certificate>();
        x509Content.add(cert);
        X509Data x509data = kif.newX509Data(x509Content);
        KeyValue keyValue = kif.newKeyValue(cert.getPublicKey());
        List info = new ArrayList();
        info.add(x509data);
        info.add(keyValue);
        KeyInfo ki = kif.newKeyInfo(info);
        /*Crear el DOMSignContext y especificar la clave privada RSA y la
ubicación en la que se insertará el elemento de firma resultante.*/
```

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true); // necesario para tener en cuenta los namespace del
documento xml
DOMSignContext dsc = new DOMSignContext(keyEntry.getPrivateKey(),

    msg.getElementsByTagNameNS("http://schemas.xmlsoap.org/soap/envelope/",
"Header").item(0));
    Element          elem          =          (Element)
msg.getElementsByTagNameNS("http://schemas.xmlsoap.org/soap/envelope/", "Body").item(0);
    //Asignar Id al elemento firmado
    elem.setAttributeNS(null, "Id", idElemento);
    Attr idAttr = elem.getAttributeNode("Id");
    elem.setIdAttributeNode(idAttr, true);
    dsc.setIdAttributeNS(elem, null, "Id");
    // Insertar el prefijo de namespace para la firma
    dsc.setDefaultNamespacePrefix("sec");
    // Generar el XMLSignature.
    XMLSignature signature = fac.newXMLSignature(si, ki);
    // Generar la firma.
    signature.sign(dsc);
    return new TransformUtils().document2String(msg);
}
```

8 Anexo V. Firmas admitidas de los servicios.

Para la validación de las firmas de la gran mayoría de los servicios se admiten los siguientes algoritmos de firmas:

- **RSA-SHA1**
- **RSA-SHA256**
- **ECDSA-SHA1**

Para la firma de tipo **XMLDSig** de los servicios de **AEAT no intermediarios**, su validación no depende del algoritmo de firma utilizado, y por tanto, no está sujeto a la restricción anterior.

9 Anexo VI. Librería PAI-LIB

Para facilitar a los desarrolladores la tarea de implementar los interceptores para firmar las peticiones se ha creado una librería llamada **PAI-LIB**.

Para descargar la librería se debe incluir la dependencia en **pom.xml** correspondiente en el proyecto.

```
<dependency>
  <groupId>es.gva.pai.webservice.firma</groupId>
  <artifactId>pai-lib-security</artifactId>
  <version>01.00.00</version>
</dependency>
```

Esta librería ofrece poder firmar mediante **XMLDSig** y **WSSecurity**, de la siguiente forma:

- **XMLDSig:**

- Se debe de crear un nuevo objeto de tipo **XmlFirmaOutInterceptor**, el cual llamara a la factoría de creación de interceptores con la función **createInterceptorXmlFirmaOut**.
- Los parámetros a pasar a la función son los siguientes:
 - **rutaKeystore**. La ruta del certificado con el que se quiere firmar la petición.
 - **keystorePassword**. La contraseña del certificado.
 - **keyAlias**. Alias del certificado.
 - **aliasPassword**. Contraseña del alias del certificado.
 - **keystoreType**. Tipo del certificado.

```
XmlFirmaOutInterceptor xmlFirmaOutInterceptor =
InterceptoresFactory.createInterceptorXmlFirmaOut(rutaKeystore, keystorePasswo
rd, keyAlias, aliasPassword, keystoreType);
```

- **WSecurity:**

- Se debe de crear un nuevo objeto de tipo **WsSecurityFirmaOutInterceptor**, el cual llamara a la factoría de creación de interceptores con la función **createInterceptorFirmaWsSecurity**.
- Los parámetros a pasar a la función son los siguientes:
 - **rutaKeystore**. La ruta del certificado con el que se quiere firmar la petición.

- **keystorePassword**. La contraseña del certificado.
- **keyAlias**. Alias del certificado.
- **aliasPassword**. Contraseña del alias del certificado.
- **keystoreType**. Tipo del certificado.

```
WsSecurityFirmaOutInterceptor wsSecurityFirmaOutInterceptor =  
InterceptoresFactory.createInterceptorFirmaWsSecurity(rutaKeystore, keystorePa  
ssword, keyAlias, aliasPassword, keystoreType);
```

La librería también ofrece la posibilidad de añadir la cabecera **IdTrazabilidad** de una forma sencilla.

Para ello se debe llamar a la factoría de interceptores y crear un nuevo objeto de tipo **IdTrazabilidadOutInterceptor** donde se llamará al método **createInterceptorIdTrazabilidad**.

El método está **sobrecargado**, es decir, existe dos tipos de implementación dependiendo de los parámetros que se le proporcionen.

- En la primera implantación se le proporcionará el **certificado** y el **código autorización** con el cual se implementará el **IdTrazabilidad**.

```
IdTrazabilidadOutInterceptor interceptor =  
InterceptoresFactory.createInterceptorIdTrazabilidad("CERTIFICADO_PRUEBA",  
"CODIGO_AUTORIZACION_PRUEBA");
```

- En la segunda implementación **únicamente** se le proporcionará el **código autorización**.

```
IdTrazabilidadOutInterceptor interceptor =  
InterceptoresFactory.createInterceptorIdTrazabilidad("CODIGO_AUTORIZACION_PRU  
EBA");
```