

# **Práctica de SOA**

---

## **Servicios SOAP usando JEE (Glassfish)**

---

### **Arquitecturas y Tecnologías del Software**

#### **Autores**

**David Lozano Jarque (NIU 1359958)**

**Carlos González Cebrecos (NIU 1212586)**

**Adrián Soria Bonilla (NIU 1360940)**

Escuela de Ingeniería, Universidad Autónoma de Barcelona

Curso 2016-2017

# Introducción

En esta práctica, aprenderemos a crear y consumir servicios SOA usando el protocolo SOAP con intercambio de mensajes en XML sobre HTTP(S) con la ayuda de las tecnologías J2EE, y en particular, *Glassfish*

## Preparación del entorno

### Instalación de Glassfish

En primer lugar, debemos instalar *Glassfish* para crear un servidor de aplicaciones de JEE donde lanzar nuestros servicios web SOAP en forma de contenedores Java.

Para ello, nos descargamos la [distribución oficial de Glassfish](#).

#### GlassFish Server Open Source Edition 4.1.2 Download

GlassFish Open Source Edition	Nightly Builds	Java EE SDK	Maven	Oracle GlassFish Server	Earlier Releases
Step 0. Prerequisite	Java EE 7 requires JDK 7 minimum. GlassFish 4.1.2 has been certified on JDK7 u80.				
Step 1. Download	Java EE 7 Web Profile ↳ glassfish-4.1.2-web.zip		Java EE 7 Full Platform ↳ glassfish-4.1.2.zip		
Step 2. Install	unzip glassfish-4.1.2*.zip This command will extract GlassFish with a preconfigured 'domain1' domain.				
Step 3. Start	glassfish4/bin/asadmin start-domain				
Step 4. Load Console	Go to <a href="http://localhost:4848">http://localhost:4848</a>				
Step 5. Check the documentation					

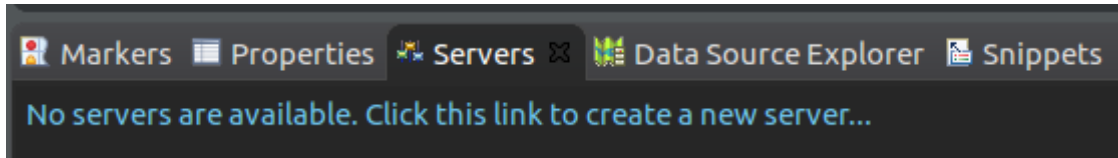
A continuación, instalaremos *Glassfish* en nuestro ordenador descomprimiendo el .zip en un directorio.

Para este tutorial usaremos una máquina con S.O. Ubuntu 16.04 LTS. Las instrucciones para otros sistemas operativos son semejantes. También damos por supuesto que tenemos Eclipse para JEE y el último JDK instalados correctamente.

```
davidlj@homeplace:/opt$  
davidlj@homeplace:/opt$ sudo cp ~/Downloads/glassfish-4.1.2.zip .  
davidlj@homeplace:/opt$ sudo unzip glassfish-4.1.2.zip  
Archive:  glassfish-4.1.2.zip  
  creating: glassfish4/  
  creating: glassfish4/bin/  
 inflating: glassfish4/bin/asadmin  
 inflating: glassfish4/bin/asadmin.bat  
 inflating: glassfish4/bin/pkg
```

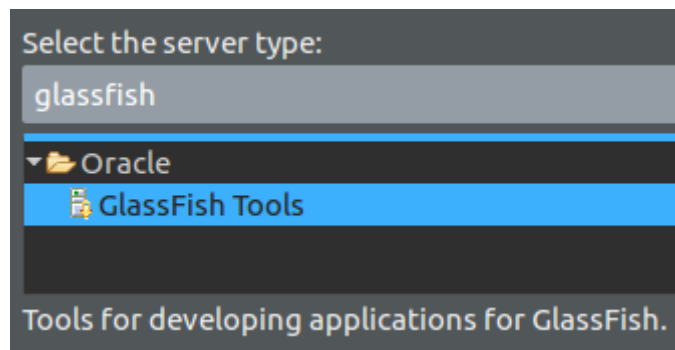
# Integración de *Glassfish* en *Eclipse*

Ahora debemos controlar la administración del servidor de *Glassfish* desde *Eclipse* para facilitar un entorno IDE completamente integrado. Para ello, en la pestaña de *Servers*, en la parte inferior de la pantalla, pulsamos sobre [...] *create a new server...*



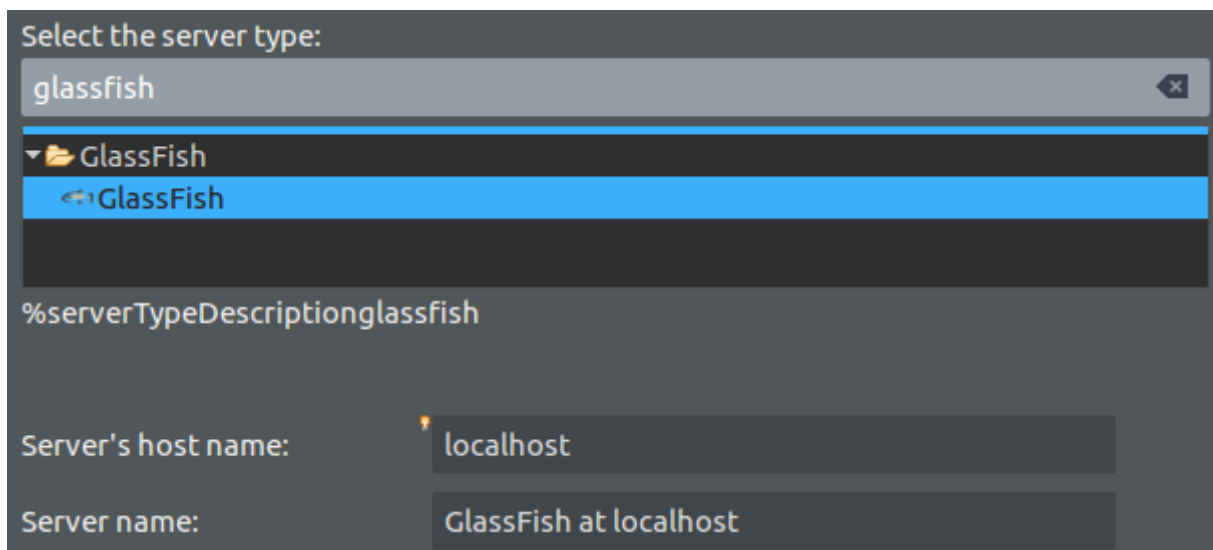
Si la pestaña de *Servers* no aparece, la podemos hacer aparecer desde el menú principal *Window -> Show View -> Other...* y buscando *servers* allí

A continuación, en el tipo de servidor seleccionamos *Glassfish Tools*. En el caso que no tengamos el conector de *Eclipse* para *Glassfish* instalado, éste se instalará automáticamente.

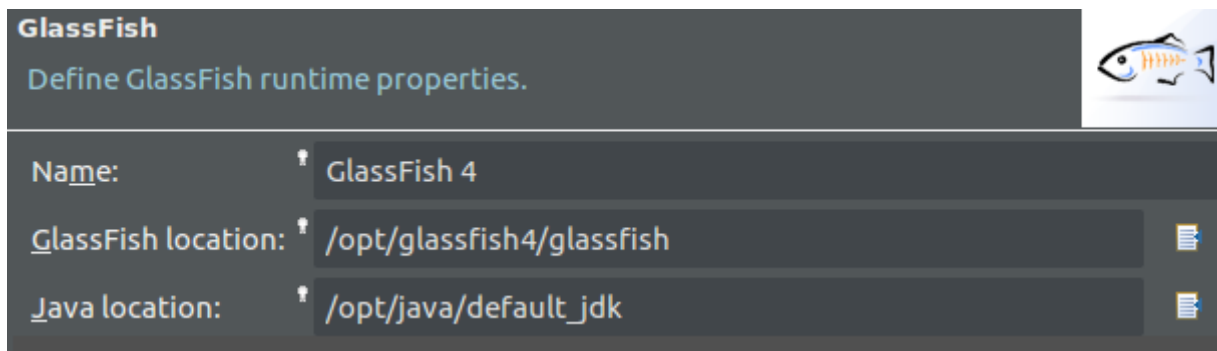


En el caso de que se cuelgue la instalación o nunca se llegue a completar, podemos instalar el complemento *Glassfish Tools* para Eclipse dentro del *Eclipse Marketplace*, en el menú *Help -> Eclipse Marketplace*

Creamos un nuevo servidor local con los parámetros por defecto (*localhost* y nombre *Glassfish at localhost*)



En este paso del tutorial, le indicamos el nombre para referenciar el servidor, la ubicación de *Glassfish* (dónde descomprimos anteriormente, la carpeta interna llamada *glassfish*) y finalmente el *JDK* a usar.



The screenshot shows the 'GlassFish' configuration window with the title 'Define GlassFish runtime properties.' and a fish icon. It contains three fields: 'Name' set to 'GlassFish 4', 'GlassFish location' set to '/opt/glassfish4/glassfish', and 'Java location' set to '/opt/java/default\_jdk'. Each field has a small upward arrow icon to its left and a document icon to its right.

Ahora debemos crear un dominio para lanzar nuestro conjunto de aplicaciones web. Crearemos un dominio llamado *ATS*. Para ello, nos situamos en el directorio `glassfish/bin` dentro del directorio de instalación de *Glassfish* y ejecutamos el comando `asadmin` para crear un dominio:

```
./asadmin create-domain ats
```

```
davidlj@homeplace:/opt/glassfish4/glassfish/bin$ ./asadmin
davidlj@homeplace:/opt/glassfish4/glassfish/bin$ ./asadmin create-domain ats
Enter admin user name [Enter to accept default "admin" / no password]>
Using default port 4848 for Admin.
Using default port 8080 for HTTP Instance.
Using default port 7676 for JMS.
Using default port 3700 for IIOP.
Using default port 8181 for HTTP_SSL.
Using default port 3820 for IIOP_SSL.
Using default port 3920 for IIOP_MUTUALAUTH.
Using default port 8686 for JMX_ADMIN.
Using default port 6666 for OSGI_SHELL.
Using default port 9009 for JAVA_DEBUGGER.
Distinguished Name of the self-signed X.509 Server Certificate is:
[CN=homeplace,OU=GlassFish,O=Oracle Corporation,L=Santa Clara,ST=California,C=US]
Distinguished Name of the self-signed X.509 Server Certificate is:
[CN=homeplace-instance,OU=GlassFish,O=Oracle Corporation,L=Santa Clara,ST=California,C=US]
Domain ats created.
Domain ats admin port is 4848.
Domain ats allows admin login as user "admin" with no password.
Command create-domain executed successfully.
```

A continuación, *Eclipse* nos pedirá la ubicación del dominio (ubicado dentro de la instalación de *Glassfish* `glassfish/domains/<dominio>`). Por defecto el usuario es `admin` y no hay contraseña.

## GlassFish

Define GlassFish Application Server properties.



Domain path:



Admin name:

Admin password:

Debug port:

☒ Preserve sessions across redeployment

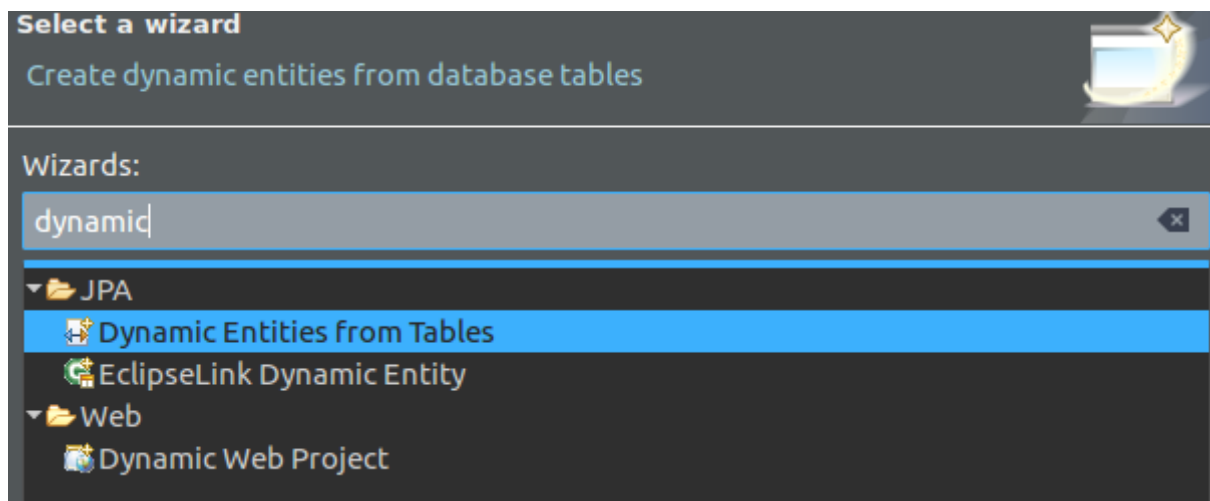
☐ Use JAR archives for deployment

# Creando un servicio SOAP y su definición WSDL

---

## Creación del proyecto para un servicio SOAP


Creamos un nuevo proyecto de *Eclipse*, del tipo *Other* y en el desplegable, seleccionamos *Dynamic Web Project* dentro de la categoría *Web*



Especificamos el nombre de proyecto, ubicación del proyecto y si hemos configurado bien el servidor, debería aparecer ya en el seleccionable *Target runtime* seleccionado por defecto.

**Dynamic Web Project**

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.



**Project name:** soap-test

**Project location**

☒ Use default location

**Location:** /home/davidlj/files/workspaces/eclipse-jee-neon/soap-test [Browse...](#)

**Target runtime**

GlassFish 4 [New Runtime...](#)

**Dynamic web module version**

3.1

**Configuration**

Default Configuration for GlassFish 4 [Modify...](#)

A good starting point for working with GlassFish 4 runtime. Additional facets can later be installed to add new functionality to the project.

**EAR membership**

☐ Add project to an EAR

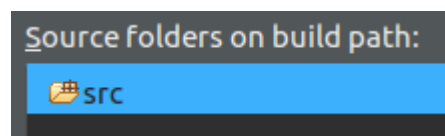
**EAR project name:** EAR [New Project...](#)

**Working sets**

☐ Add project to working sets [New...](#)

**Working sets:** [Select...](#)

En la siguiente pantalla, vemos las carpetas a compilar. La carpeta `src` se compilará por defecto, que es más que suficiente.



Finalmente, especificamos el directorio donde se guardarán los recursos web estáticos (CSS, HTML, JS,...). Pulsamos sobre *Generate web.xml deployment descriptor* para generar un descriptor de contenidos web a publicar por defecto (los situados en *WebContent*)

**Web Module**  
Configure web module settings.

Context root: soap-test

Content directory: WebContent

☒ Generate web.xml deployment descriptor

## Desarrollo del servicio SOAP

Creamos una nueva clase que contendrá un servicio SOAP, llamado *TestWebService*

**Java Class**  
Create a new Java class.

Source folder: soap-test/src Browse...

Package: codes.uab.ats.soap\_test Browse...

☐ Enclosing type: Browse...

---

Name: TestWebService

Modifiers: ☒ public ☒ package ☐ private ☐ protected  
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

En dicha clase, indicamos las anotaciones `@WebService` para indicar que es un servicio web y `@WebMethod` en el método `public String sayHello(String name)` para indicar que es un servicio SOAP a ofrecer que acepta un parámetro (nombre) y devuelve otro parámetro saludando a dicho nombre (ambos de tipo *String*)



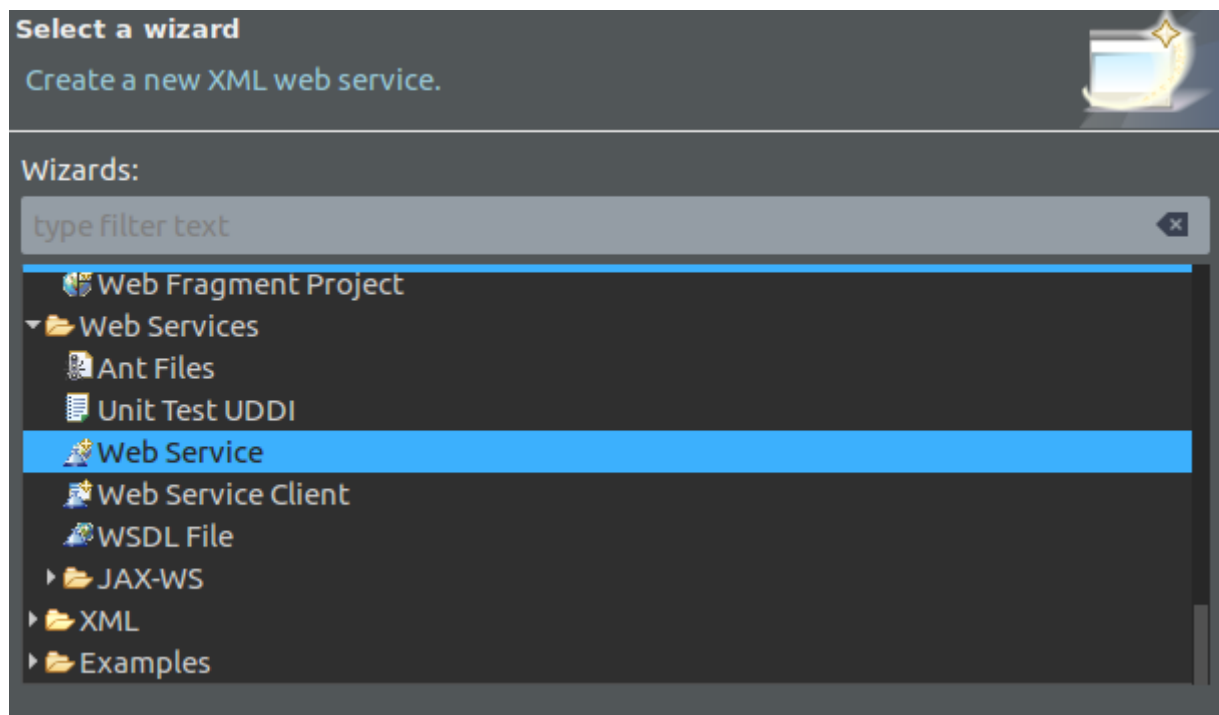
```

1 package codes.uab.ats.soap_test;
2
3 import javax.jws.WebMethod;
4 import javax.jws.WebService;
5
6 @WebService
7 public class TestWebService {
8
9     @WebMethod
10    public String sayHello(String name) {
11        return "Hello " + name;
12    }
13 }
14

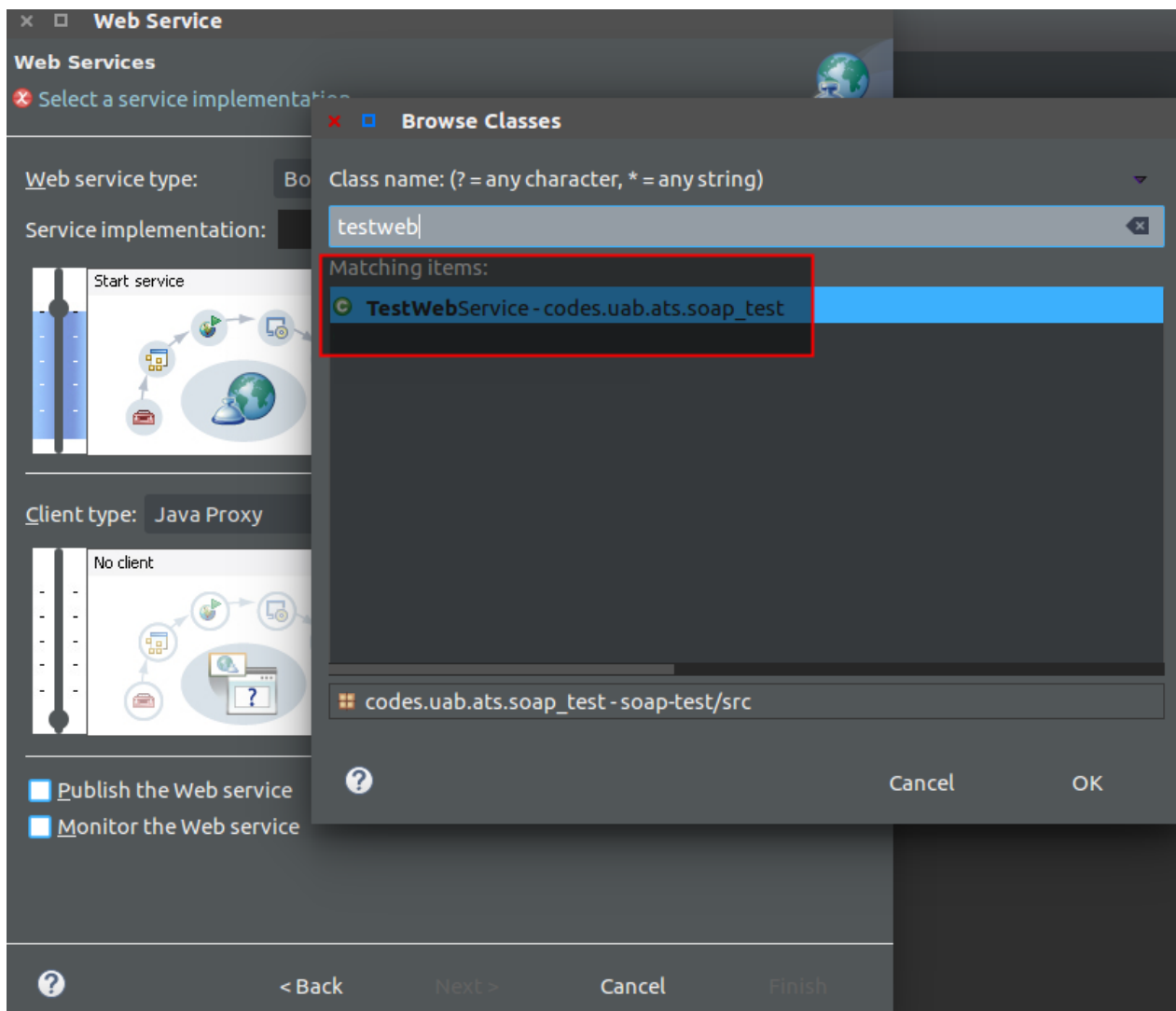
```

## Publicación de la descripción del servicio SOAP via WSDL

Generamos un fichero *WSDL* para que clientes puedan consumir nuestro servicio con la operación *sayHello*. Para ello generamos un *WSDL* que describa este simple *webservice*. Pulsamos clic derecho en nuestro proyecto y *New -> Other...*, allí seleccionamos *Web Service*



En este paso indicamos la clase creada anteriormente como clase que ofrecerá el servicio web y seleccionamos los niveles de configuración del servicio web (seleccionamos que el servicio se comience a ejecutar y no use proxy)



Finalmente, seleccionamos las operaciones a exportar en el WSDL que describe el servicio (en nuestro caso sólo tenemos la operación *sayHello*) y finalizamos el asistente

**Web Service Java Bean Identity**  
Configure the Java bean as a Web service.

WSDL file:

Methods

- ✓ sayHello(java.lang.String)

Select All   Deselect All

Style and use

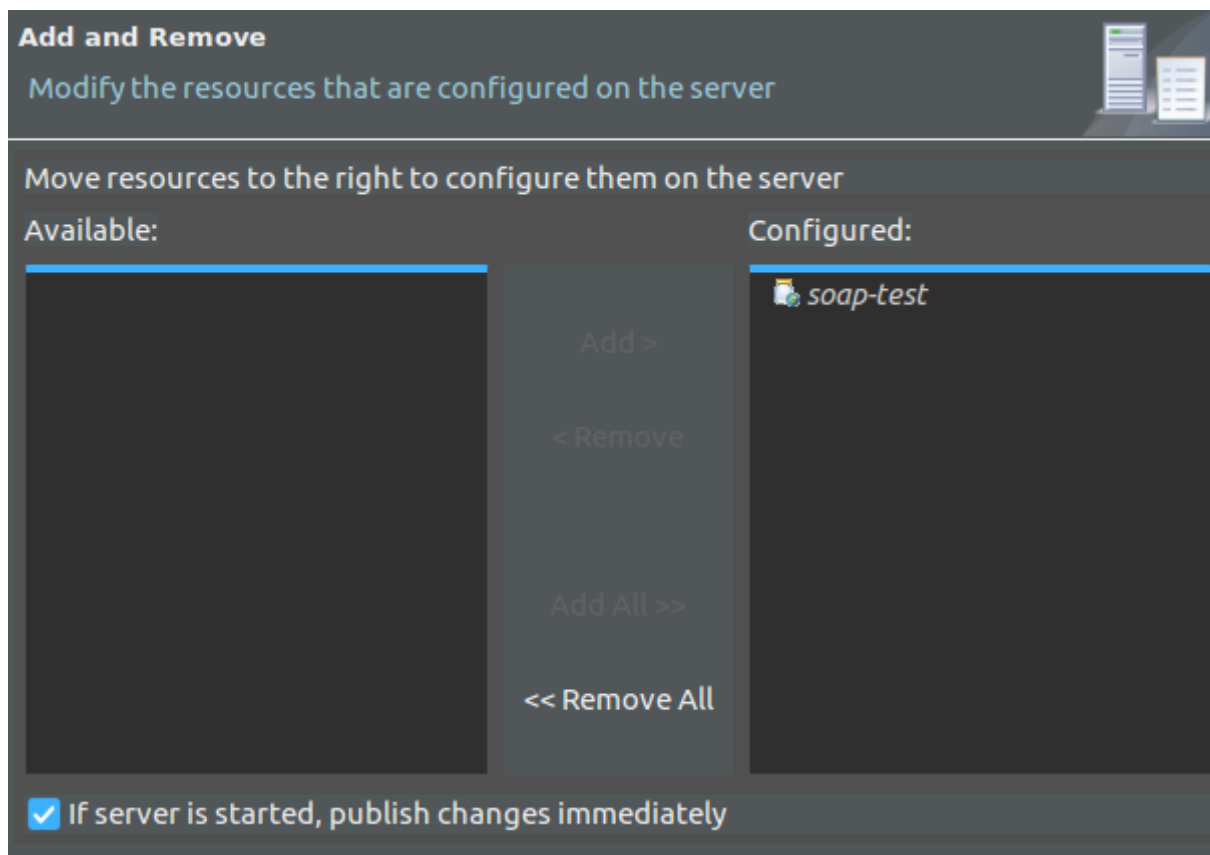
- ☒ document/literal (wrapped)
- ☐ document/literal
- ☐ RPC/encoded

☐ Define custom mapping for package to namespace.

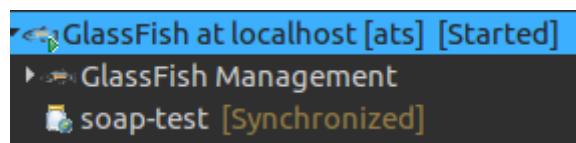
?   < Back   Next >   Cancel   Finish

## Puesta en marcha del servidor *Glassfish* con el servicio SOAP

Para lanzar el servidor *Glassfish* con nuestra aplicación publicada, clicamos en la pestaña de servidores clic derecho y pulsamos en *Add and Remove*.... Seleccionamos nuestro proyecto y lo añadimos a la lista de *Configured* pulsando en *Add*.



Finalmente, iniciamos el servidor haciendo clic derecho en el servidor y pulsando en *Start*. El resultado será parecido al siguiente (con las indicaciones *Started* y *Synchronized*)



## Administración del servidor *Glassfish*

Para entrar en el panel de administración web de *Glassfish*, abrimos un navegador y escribimos la siguiente URL:

<http://localhost:4848/>

Allí podremos ver nuestra aplicación publicada, en le menú de *Common Tasks*, dentro del desplegable *Applications*. Clicamos en ella para ver más detalles

Common Tasks

- Domain
  - server (Admin Server)
- Clusters
- Standalone Instances
- Nodes
- Applications
  - soap-test
- Lifecycle Modules
- Monitoring Data
- Resources
  - Concurrent Resources
  - Connectors
  - JDBC
  - JMS Resources
  - JNDI
  - JavaMail Sessions
  - Resource Adapter Configs
- Configurations
  - default-config
  - server-config
- Update Tool

GeneralDescriptor

Edit Application

Modify an existing application or module.

Name:

soap-test

Status:

☒ Enabled

Virtual Servers:

server

Associates an Internet domain name with a physical host.

Context Root:

/soap-test

Path relative to server's base URL.

Implicit CDI

☒ Enabled

Implicit discovery of CDI beans

Location:

\${com.sun.aas.instanceRootURI}/eclipseApps/soap-test

Deployment Order:

100

A number that determines the loading order of the modules.

Libraries:

Description:

Modules and Components (6)

Module Name	Engines
soap-test	[web, webservices]
soap-test	
soap-test	
soap-test	
soap-test	
soap-test	

Ahora, buscaremos nuestra URL para obtener el WSDL y consumir nuestro servicio. Podemos verla si clicamos en *View Endpoint* en el *Endpoint* de la tabla de módulos y componentes

Modules and Components (6)					
Module Name	Engines	Component Name	Type	Action	
soap-test	[web, webservices]	-----	-----	Launch	
soap-test		default	Servlet		
soap-test		AdminServlet	Servlet		
soap-test		jsp	Servlet		
soap-test		AxisServlet	Servlet		
soap-test		TestWebService	Servlet	View Endpoint	

Allí encontraremos la URL para nuestro WSDL. Si pulsamos en él, nos dejará seleccionar la URL segura (HTTPS puerto 8081) e insegura (HTTP 8080) para obtener el WSDL. Seleccionaremos la opción insegura puesto que es un caso de desarrollo / test

# Web Service Endpoint Information

View details about a web service endpoint.

<b>Application Name:</b>	soap-test
<b>Tester:</b>	/soap-test/TestWebServiceService?Tester
<b>WSDL:</b>	<a href="/soap-test/TestWebServiceService?wsdl">/soap-test/TestWebServiceService?wsdl</a>
<b>Endpoint Name:</b>	TestWebService
<b>Service Name:</b>	TestWebServiceService
<b>Port Name:</b>	TestWebServicePort
<b>Deployment Type:</b>	109
<b>Implementation Type:</b>	SERVLET
<b>Implementation Class Name:</b>	codes.uab.ats.soap_test.TestWebService
<b>Endpoint Address URI:</b>	/soap-test/TestWebServiceService
<b>Namespace:</b>	http://soap_test.ats.uab.codes/

Para aplicaciones en producción, podemos obtener un certificado gratis para nuestro dominio usando [Let's Encrypt](#) y configurarlo en *Glassfish* a través de esta misma consola de administración con la ayuda de la herramienta de [Java KeyTool](#)

Finalmente, visualizamos el XML que describe en lenguaje WSDL nuestro servicio SOAP.

```
<!-- Published by JAX-WS RI (http://jax-ws.java.net). RI's version is Metro/2.3.2-b608 (trunk-7979; 2015-01-21T12:50:19+0000) JAXWS-RI/2.2.11-b150120.1832 JAXWS-API/2.2.12 JAXB-RI/2.2.12 -->
<!-- Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is Metro/2.3.2-b608 (trunk-7979; 2015-01-21T12:50:19+0000) JAXWS-RI/2.2.11-b150120.1832 JAXWS-API/2.2.12 JAXB-RI/2.2.12 -->
<?xml version='1.0' encoding='UTF-8'?>
<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy" xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:soap="http://schemas.xmlsoap.org/soap/"
xmlns:tns="http://soap_test.ats.uab.codes/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://soap_test.ats.uab.codes/"
name="TestWebServiceService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://soap_test.ats.uab.codes/" schemaLocation="http://localhost:8080/soap-test/TestWebServiceService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="TestWebService">
    <operation name="sayHello">
      <input wsam:Action="http://soap_test.ats.uab.codes/TestWebService/sayHelloRequest" message="tns:sayHello"/>
      <output wsam:Action="http://soap_test.ats.uab.codes/TestWebService/sayHelloResponse" message="tns:sayHelloResponse"/>
    </operation>
  </portType>
  <binding name="TestWebServicePortBinding" type="tns:TestWebService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="TestWebServiceService">
    <port name="TestWebServicePort" binding="tns:TestWebServicePortBinding">
      <soap:address location="http://localhost:8080/soap-test/TestWebServiceService"/>
    </port>
  </service>
</definitions>
```

También podemos probar a usar nuestro servicio con el tester integrado cuya URL aparece en la penúltima figura. Aquí probamos nuestro servicio con la operación *sayHello* a un tal Francisco.

# TestWebServiceService Web Service Tester

This form will allow you to test your web service implementation ([WSDL File](#))

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

## Methods :

public abstract java.lang.String codes.uab.ats.soap\_test.TestWebService.sayHello(java.lang.String)

sayHello (Paco Paquito Poco Paquete)

Podemos ver que, en efecto, funciona tal y cómo esperábamos

## sayHello Method invocation

### Method parameter(s)

Type	Value
java.lang.String	Paco Paquito Poco Paquete

### Method returned

java.lang.String : "Hello Paco Paquito Poco Paquete"

### SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <S:Body>
    <ns2:sayHello xmlns:ns2="http://soap_test.ats.uab.codes/">
      <arg0>Paco Paquito Poco Paquete</arg0>
    </ns2:sayHello>
  </S:Body>
</S:Envelope>
```

### SOAP Response

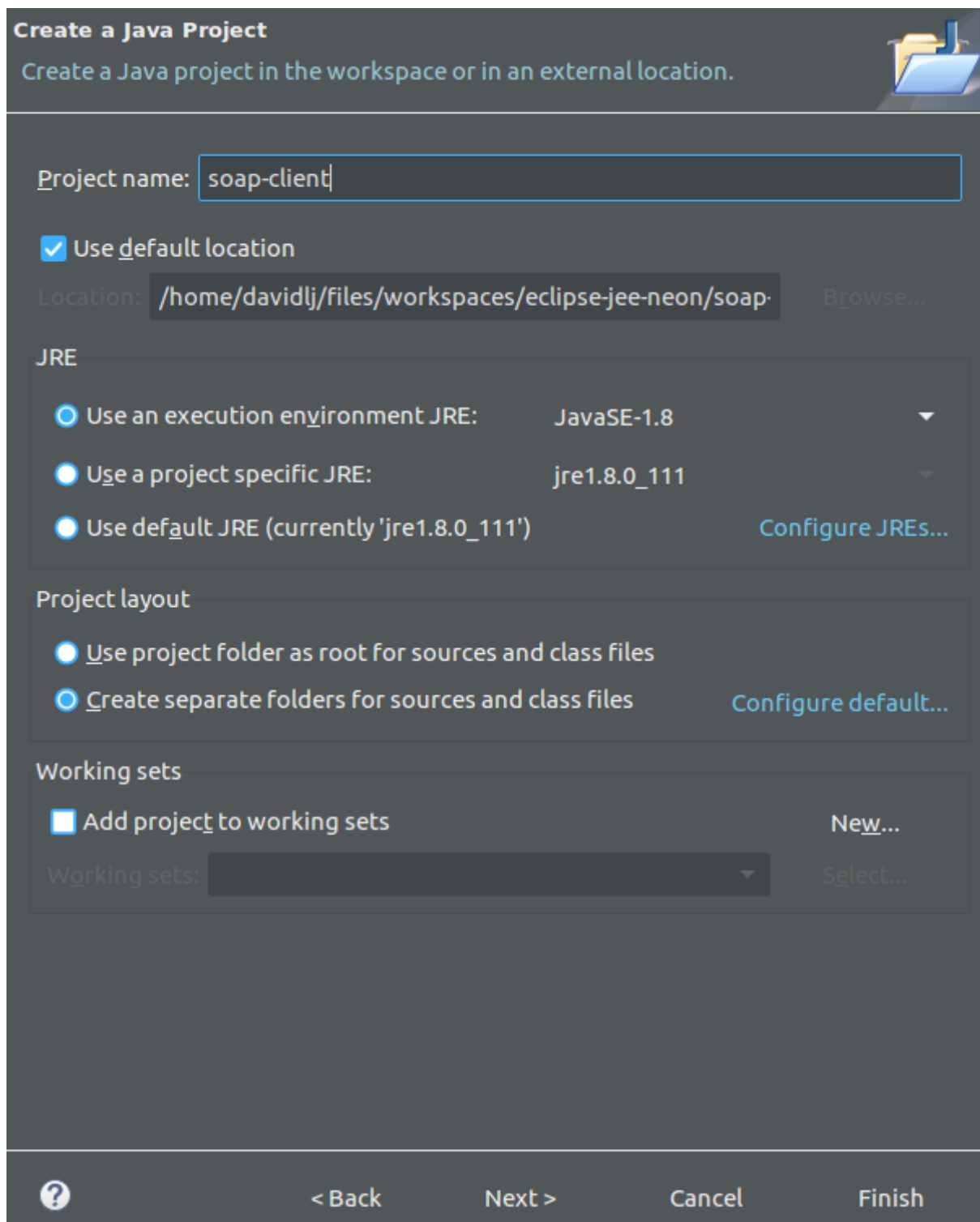
```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <S:Body>
    <ns2:sayHelloResponse xmlns:ns2="http://soap_test.ats.uab.codes/">
      <return>Hello Paco Paquito Poco Paquete</return>
    </ns2:sayHelloResponse>
  </S:Body>
</S:Envelope>
```

# Consumición de un servicio SOAP usando su WSDL

---

## Creación del proyecto consumidor de nuestro servicio SOAP

En primer lugar, creamos un proyecto básico de Java usando Eclipse:



The screenshot shows the 'Create a Java Project' dialog box in Eclipse. The title bar says 'Create a Java Project' and the subtitle says 'Create a Java project in the workspace or in an external location.' The dialog is divided into several sections:

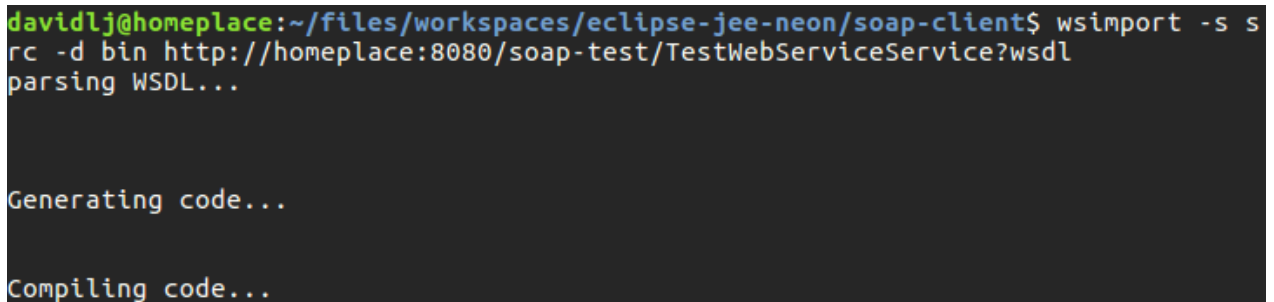
- Project name:** A text field containing 'soap-client'.
- Location:** A text field containing '/home/davidlj/files/workspaces/eclipse-jee-neon/soap'. There is a 'Browse...' button to the right.
- JRE:** A section with three radio buttons:
  - ☒ Use an execution environment JRE: JavaSE-1.8 (with a dropdown arrow)
  - ☐ Use a project specific JRE: jre1.8.0\_111 (with a dropdown arrow)
  - ☐ Use default JRE (currently 'jre1.8.0\_111')There is a 'Configure JREs...' link to the right of the third option.
- Project layout:** A section with two radio buttons:
  - ☐ Use project folder as root for sources and class files
  - ☒ Create separate folders for sources and class filesThere is a 'Configure default...' link to the right of the second option.
- Working sets:** A section with a checkbox 'Add project to working sets' which is unchecked. To the right is a 'New...' button. Below the checkbox is a 'Working sets:' label followed by a dropdown menu and a 'Select...' button.

At the bottom of the dialog, there is a navigation bar with a help icon (?), '< Back', 'Next >', 'Cancel', and 'Finish' buttons.



Una vez creado, abrimos una terminal, nos situamos dentro del directorio del nuevo proyecto y ejecutamos el comando `wsimport` ofrecido por *Java JDK* para generar código automático para consumir un SOAP dado un WSDL.

```
wsimport -s src -d bin http://localhost:8080/soap-test/TestWebServicesService?wsdl
```



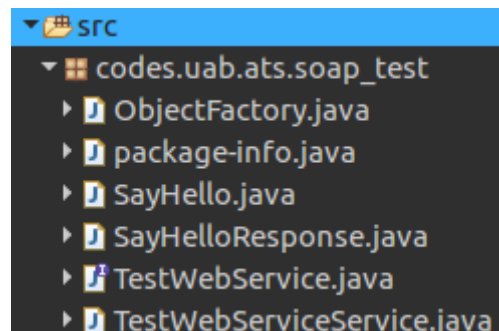
```
davidlj@homeplace:~/files/workspaces/eclipse-jee-neon/soap-client$ wsimport -s src -d bin http://homeplace:8080/soap-test/TestWebServiceService?wsdl
parsing WSDL...

Generating code...

Compiling code...
```

Si no nos reconoce la herramienta `wsimport`, nos tenemos que asegurar que el directorio `bin` del *JDK* (`$JAVA_HOME`) se encuentra en la variable `PATH` del entorno.

En el proyecto de *Eclipse*, refrescamos la carpeta `src`, clicando con el botón derecho encima de ella y pulsando en *Refresh*, veremos que se ha generado código para consumir el servicio web SOAP definido por el WSDL introducido en `wsimport`



Creamos una clase cliente que consumirá el servicio SOAP usando el código generado. Marcamos `public static void main(String[] args)` para generar un método principal ejecutable al llamar nuestra aplicación.

**Java Class**  
Create a new Java class.

Source folder: soap-client/src Browse...

Package: codes.uab.ats.soap\_test Browse...

☐ Enclosing type: Browse...

---

Name: TestClient

Modifiers: ☒ public ☒ package ☐ private ☐ protected  
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?

☒ public static void main(String[] args)  
☐ Constructors from superclass  
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))  
☐ Generate comments

? Cancel Finish

Introducimos el siguiente código en la clase para consumir el servicio. Usamos las clases generadas por `wsimport` para consumir el servicio de forma rápida y efectiva.

```
package codes.uab.ats.soap_test;

public class TestClient {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        TestWebServiceService service = new TestWebServiceService();
        TestWebService port = service.getTestWebServicePort();
        System.out.println(port.sayHello("Paco Paquito Poco Paquete"));
    }

}
```

Creamos una nueva `Run configuration` en *Eclipse* del tipo *Java Application* para compilar y ejecutar el cliente y comprobamos que la respuesta del *webservice* es correcta.

```
<terminated> TestClient [Java Applica  
Hello Paco Paquito Poco Paquete
```

## Monitorizando los mensajes SOAP via HTTP

A continuación vamos a inspeccionar los mensajes que se intercambian el cliente que consume el servicio SOAP y el servidor que lo ofrece. Para ello, en primer lugar, redirigiremos las llamadas para obtener el WSDL hacia el puerto 8081. Esto es porque en el puerto 8081 instalaremos un sniffer que nos ayudará a ver las peticiones que ejecuta (que éste redirigirá al puerto 8080 para continuar con el flujo habitual de comunicaciones).

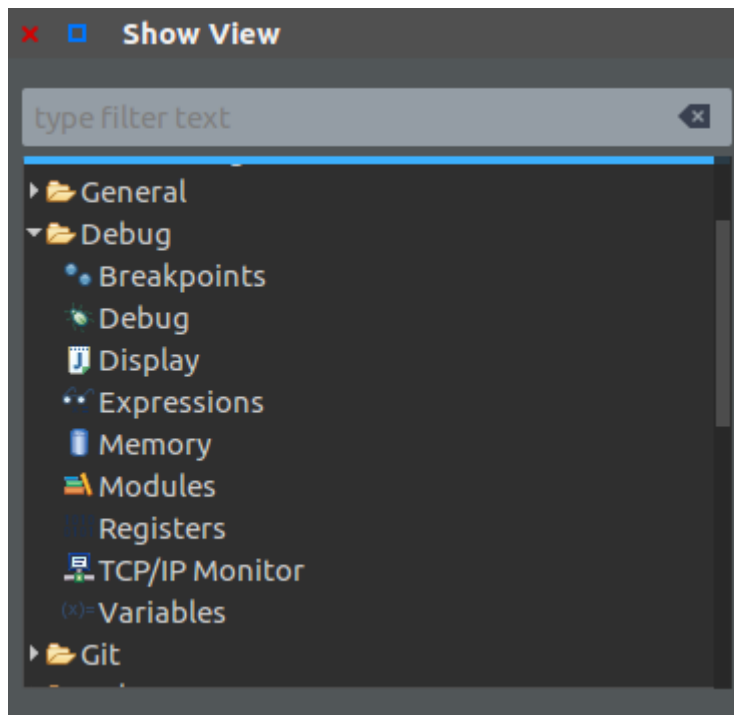
Realizamos los cambios en el fichero `TestWebServiceService.java`

```
import java.net.MalformedURLException;

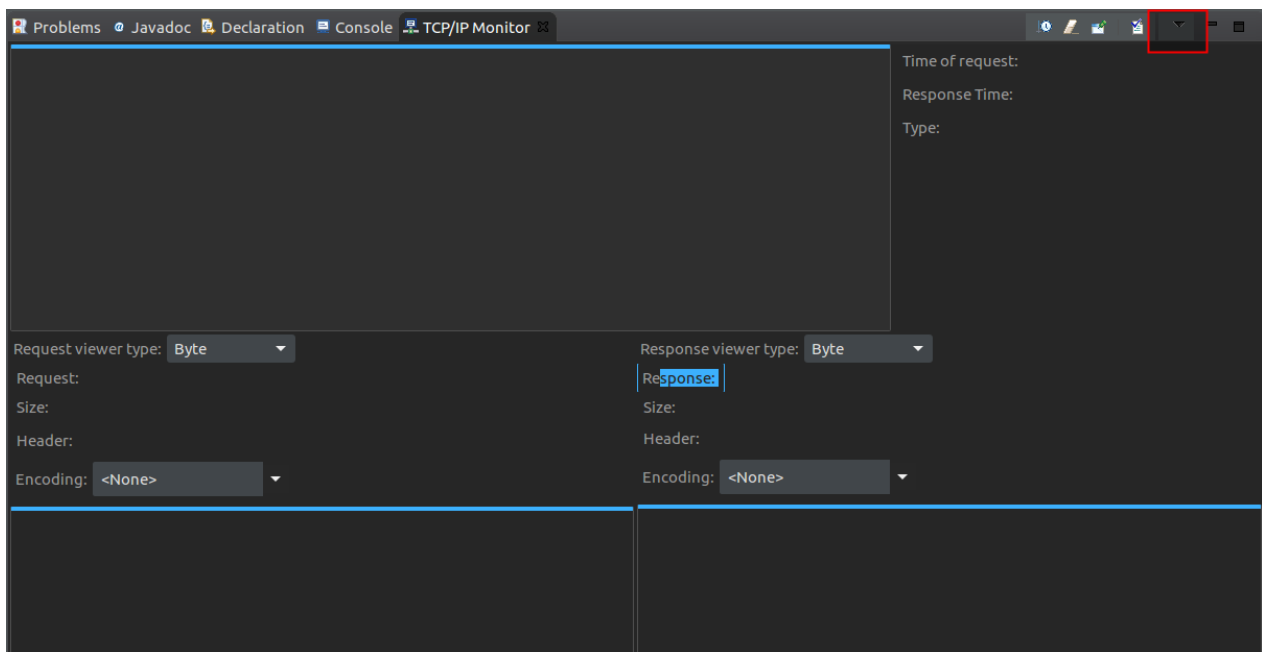
/**
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.2.9-b130926.1035
 * Generated source version: 2.2
 */
@WebServiceClient(name = "TestWebServiceService", targetNamespace = "http://homeplace:8081/soap-test")
public class TestWebServiceService
    extends Service
{
    private final static URL TESTWEBSERVICESERVICE_WSDL_LOCATION;
    private final static WebServiceException TESTWEBSERVICESERVICE_EXCEPTION;
    private final static QName TESTWEBSERVICESERVICE_QNAME;

    static {
        URL url = null;
        WebServiceException e = null;
        try {
            url = new URL("http://homeplace:8081/soap-test");
        } catch (MalformedURLException ex) {
            e = new WebServiceException(ex);
        }
        TESTWEBSERVICESERVICE_WSDL_LOCATION = url;
        TESTWEBSERVICESERVICE_EXCEPTION = e;
    }
}
```

Ahora, habilitamos el *sniffer*, llamado *TCP/IP Monitor*, pulsando sobre *Window -> Show View -> Other...* en el menú principal y allí buscando *TCP/IP Monitor* en la sección *Debug* del desplegable.



En la nueva pestaña en la sección inferior de la ventana de *Eclipse* añadimos un nuevo monitor pulsando en la flecha de más opciones en la esquina superior derecha (ver siguiente imagen) y pulsando en `Properties`



Allí, en la nueva ventana, pulsamos *Add* para añadir un nuevo monitor y lo definimos para que redirija las peticiones que recibirá en su puerto (*Local monitoring port*) a la misma máquina (*localhost*) en el puerto *Port* (8080). Seleccionamos el tipo *HTTP* puesto que no nos interesa el bajo nivel de *TCP/IP*.

**New Monitor**

Local monitoring port: 8081

Monitor

Host name: homestead

Port: 8080

Type: HTTP

Timeout (in milliseconds): 0

☐ Start monitor automatically

Cancel OK

Añadimos el monitor y hacemos clic en *Start* para comenzar el monitor.

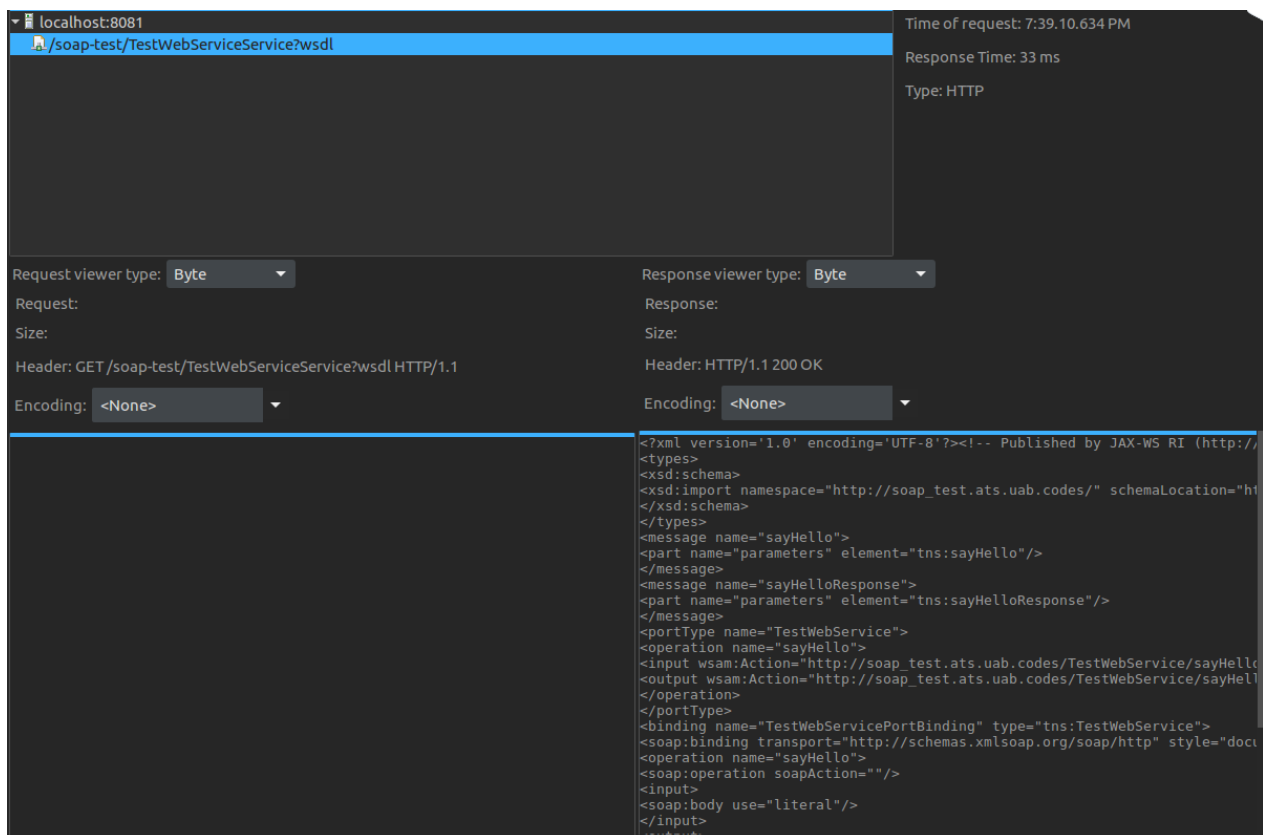
☒ Show the TCP/IP Monitor view when there is activity

TCP/IP Monitors:

Status	Host name	Type	Local Po	Auto-star
Started	homestead:8080	HTTP	8081	No

Add...  
Edit...  
Remove  
Start  
Stop  
Columns...

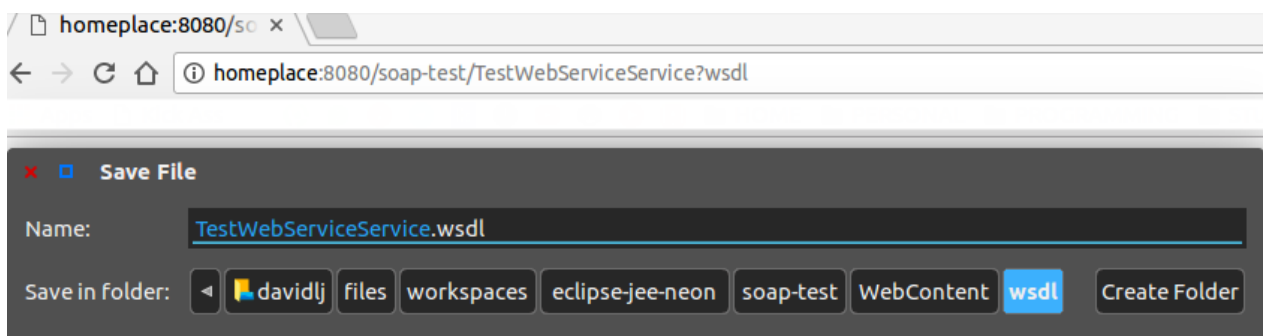
A continuación, ejecutamos el cliente y veremos que la primera y única petición que aparece es la petición para obtener el WSDL ( `GET /soap-test/TestWebServiceService?wsdl` ), en cuya respuesta el servidor sirve el WSDL que vimos anteriormente. Los mensajes de *request* y *response* de la operación *sayHello* no aparecen



Esto es puesto que al obtener el WSDL, el WSDL indica el puerto 8080 para usar el servicio, por lo que la llamada a la operación *sayHello* la realiza a dicho puerto y el sniffer no intercepta la petición. Para poder realizar la inspección de las operaciones, debemos cambiar el puerto del WSDL.

Dado que el WSDL en dicha URL se genera dinámicamente, obtendremos el WSDL, lo guardaremos como recurso estático con el puerto cambiado y redirigiremos las llamadas al nuevo WSDL con el puerto cambiado indicando que los servicios se encuentran en el puerto 8081

Volvemos a visitar el WSDL con el navegador y guardamos el fichero en la ubicación `WebContent\wsdl\TestWebServiceService.wsdl`.



Lo abrimos y cambiamos el puerto de 8080 a 8081 en el fichero que acabamos de guardar.

```

<service name="TestWebServiceService">
  <port name="TestWebServicePort" binding="tns:TestWebServicePortBinding">
    <soap:address location="http://localhost:8081/soap-test/TestWebServiceService"/>
  </port>
</service>

```

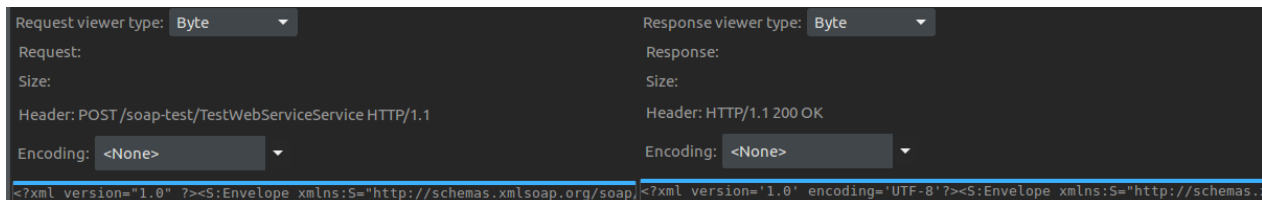
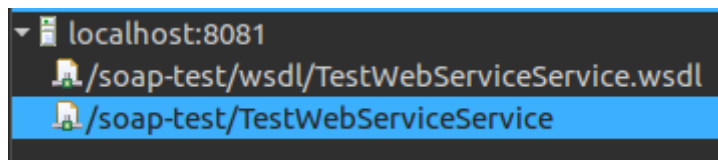
Publicamos los cambios en *Glassfish* clicando con el botón derecho sobre el servidor y haciendo clic en *Publish*.

Ahora modificamos la URL del WSDL en la clase `TestWebServiceService.java` para que en vez de visitar el WSDL generado dinámicamente, visite el estático que acabamos de guardar, substituyendo la URL de la cuál cambiamos el puerto de

`http://localhost:8081/soap-test/TestWebServiceService?wsdl` hacia

`http://localhost:8081/soap-test/wsdl/TestWebServiceService.wsdl`

Ejecutamos de nuevo el cliente SOAP. Ahora ya vemos que el sniffer puede coger las dos peticiones, la del nuevo WSDL y la de la operación *sayHello*. En dicha operación, se hace un *POST* con el *Envelope SOAP* que contiene los parámetros a solicitar para la operación y la operación a solicitar y la respuesta contiene el *Envelope SOAP* con el objeto respuesta a la operación



Visualizamos los resultados en un editor de texto

```

1  <!-- REQUEST -->
2  <?xml version="1.0" ?>
3  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
4  |   <S:Body>
5  |   |   <ns2:sayHello xmlns:ns2="http://soap_test.ats.uab.codes/">
6  |   |   |   <arg0>Paco Paquito Poco Paquete</arg0>
7  |   |   </ns2:sayHello>
8  |   </S:Body>
9  </S:Envelope>
10
11 <!-- RESPONSE -->
12 <?xml version='1.0' encoding='UTF-8'?>
13 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
14 |   <S:Body>
15 |   |   <ns2:sayHelloResponse xmlns:ns2="http://soap_test.ats.uab.codes/">
16 |   |   |   <return>Hello Paco Paquito Poco Paquete</return>
17 |   |   </ns2:sayHelloResponse>
18 |   </S:Body>
19 </S:Envelope>

```

## Detalle de los mensajes SOAP

En función de la versión de SOAP que se esté usando en el momento, podemos encontrar los mensajes SOAP situados dentro del tag `<S:Envelope>`

`<SOAP:Envelope>` o bien `<S:env>` .

En la petición vemos que en el cuerpo del mensaje soap se solicita la operación

`sayHello` , con el argumento `Paco Paquito Poco Paquete` y en la respuesta obtenemos un objeto `sayHelloResponse` con valor de retorno `Hello Paco Paquito Poco Paquete`

## Consumición de un servicio SOAP externo

A continuación crearemos un tercer proyecto para consumir un servicio SOAP público existente, usando las instrucciones para generar nuestro último cliente consumidor de un servicio SOAP.

### El servicio SOAP

Usaremos un servicio SOAP proporcionado por [WebServiceX](#) que obtiene la localización geográfica de una dirección IP. Vemos que nos proporcionan el *endpoint* donde se encuentra el WSDL.



# GeoIPService Detail

GeoIPService enables you to easily look up countries by IP address / Context

## Endpoint

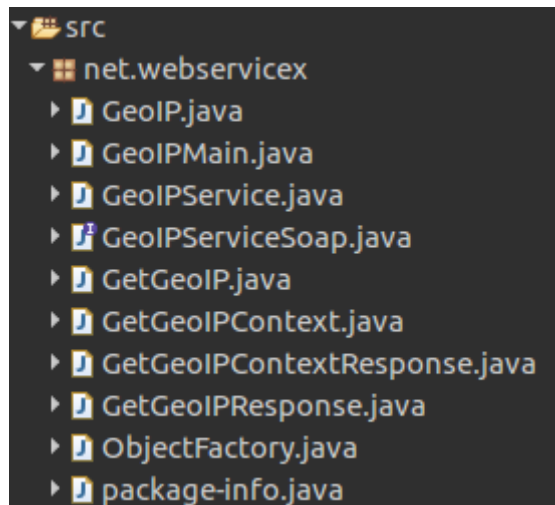
<http://www.webserviceex.net/geoipservice.asmx?WSDL>

Creamos un nuevo proyecto de Java como en el apartado anterior y usamos el comando `wsimport` dentro de la carpeta de este nuevo proyecto para generar el código que consumirá el servicio.

El comando es el siguiente:

```
wsimport -s src -d bin http://www.webserviceex.net/geoipservice.asmx?WSDL
```

Refrescamos la carpeta `src` igual que en el anterior apartado y vemos que ya tenemos el código para consumir el servicio.



Creamos una nueva clase y llamámos al servicio, siendo la dirección IP un parámetro del cliente, que nos imprimirá el país y su código identificativo dada una IP que geolocaliza

```

package net.webservicex;

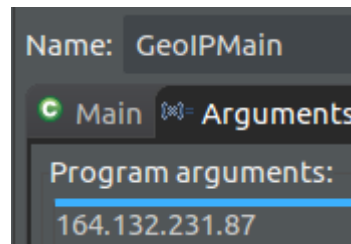
public class GeoIPMain {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        GeoIPService service = new GeoIPService();
        GeoIPServiceSoap port = service.getGeoIPServiceSoap();
        GeoIP response = port.getGeoIP(args[0]);
        System.out.println(
            String.format("The IP location country is %s and its code is %s",
                response.countryName, response.countryCode));
    }

}

```

A continuación creamos una nueva `Run configuration`, del tipo *Java Application* y indicamos en los parámetros la IP a geolocalizar



Ejecutamos el cliente y nos devuelve la geolocalización de la IP pasada como argumento.

```

<terminated> GeoIPMain [Java Application] /opt/java/jre1.8.0_111/bin/jav
The IP location country is European Union and its code is EU

```

## Repositorio

El código del *webservice* SOAP que proporciona la operación `sayHello` se encuentra en el siguiente repositorio de *GitHub*:

<https://github.com/uab-projects/ats-soap-test>

El código del consumidor de dicho *webservice* se encuentra en el siguiente repositorio:

<https://github.com/uab-projects/ats-soap-client>

Finalmente, el código del consumidor del servicio de geolocalización de IPs mediante SOAP se encuentra en el siguiente repositorio:

<https://github.com/uab-projects/ats-geoip-client>

Esta documentación está presente en formato *Markdown*, *PDF* y *HTML* en:

<https://github.com/uab-projects/ats-soap-test>

En el directorio `docs` , y visible online en la siguiente URL

<https://uab.codes/ats-soap-test/>