Sistemas Distribuidos e Internet
Servicios Web SOAP con Spring Boot
Sesión- 12
Curso 2017/ 2018

1 Introducción

En este apartado vamos a desarrollar un servidor y un cliente de ***servicios web basados en SOAP***, usando Spring boot. En primer lugar, desarrollaremos un servidor que va a exponer datos de las notas de las asignaturas de los alumnos. En segundo lugar, construiremos un cliente que obtenga datos del servicio web anterior, a partir del WSDL generado.

1.1 Desarrollando el servicio Web SOAP

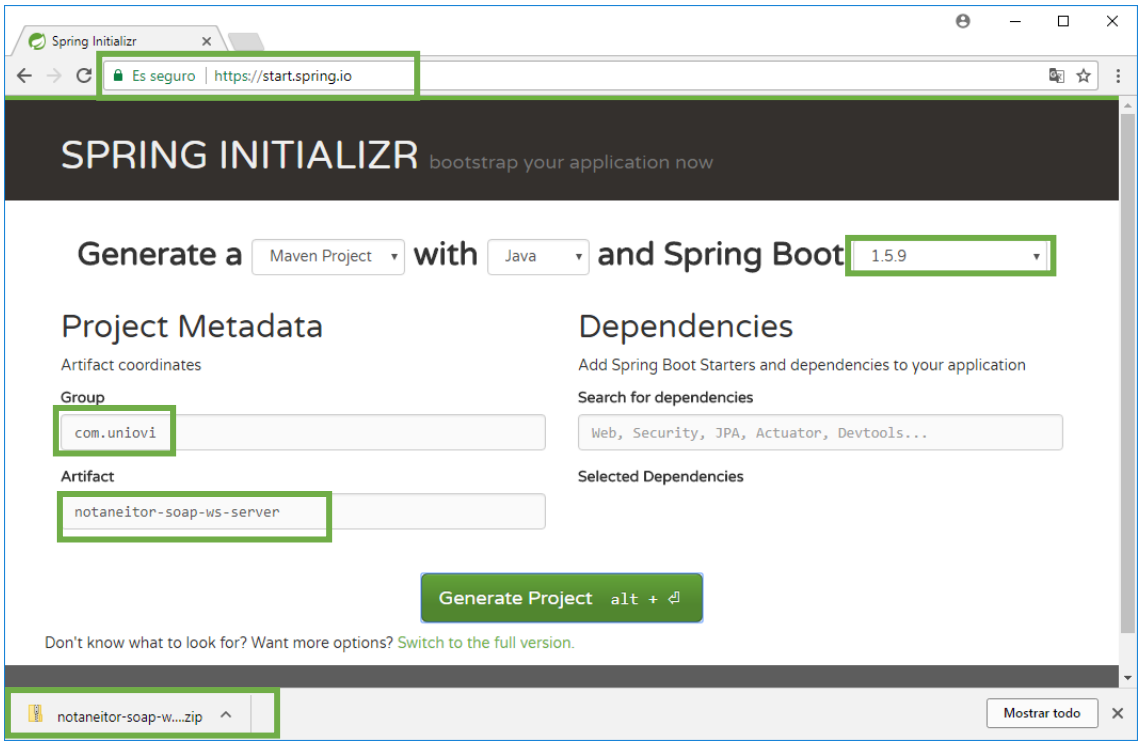
1.1.1 Creación del proyecto

Un proyecto Spring Boot es básicamente un proyecto Java Maven, con al menos la dependencia a la librería "spring-boot-starter-web". Las tres alternativas más habituales para crear un nuevo proyecto Spring son:

1. Crear un proyecto Maven desde el entorno e incluir la dependencia en el pow.xml (maven)
2. Generar una plantilla a través del inicializador de Spring: https://start.spring.io/ . Seleccionando al menos la dependencia "Web" (Full-stack web development with Tomcat and Spring MVC)
3. Crear un proyecto "Spring Starter Project" e incluir la dependencia web desde el asistente.

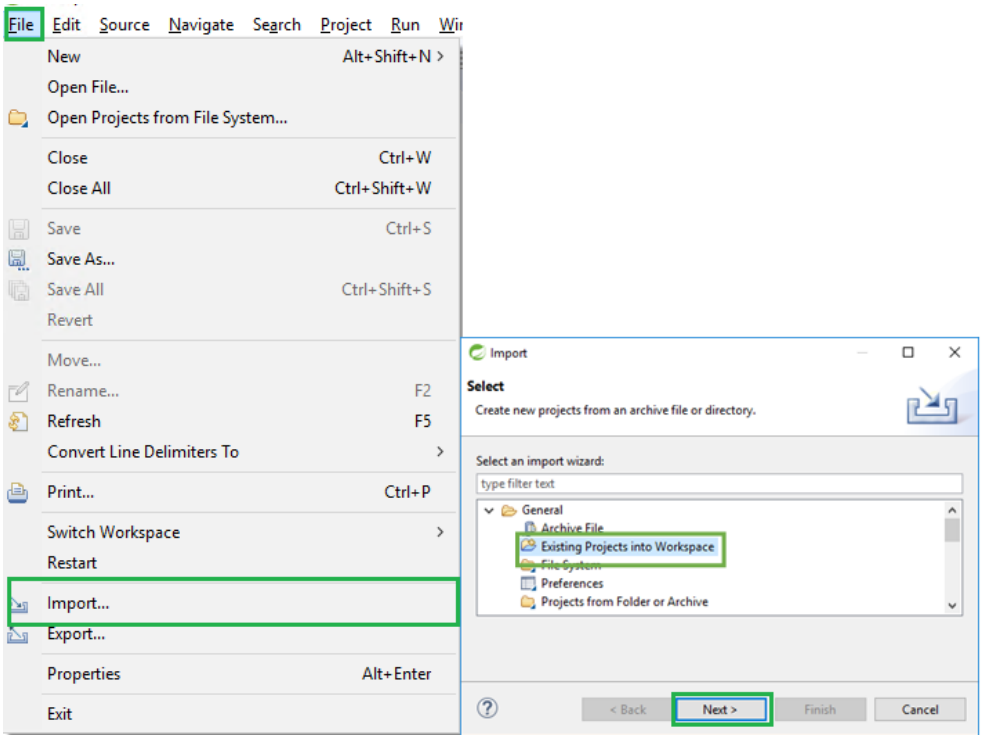
Para desarrollar el servidor vamos a optar por la opción 2 aunque las otras serian igualmente válidas.

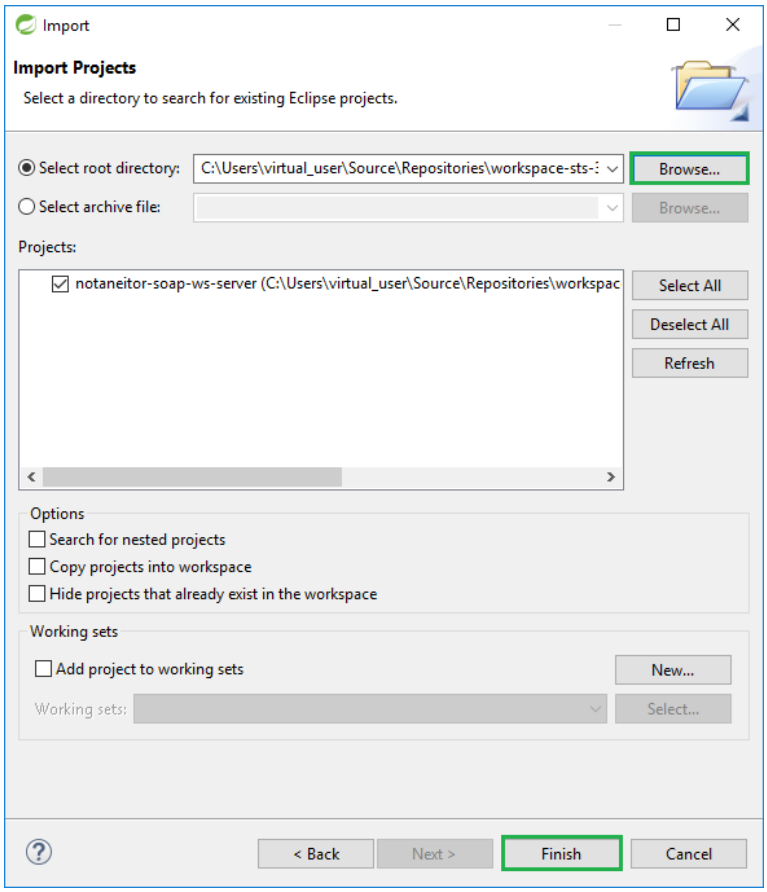
Utilizando un navegador web vamos la página web https://start.spring.io/ y generamos un nuevo proyecto con los datos que se muestran a continuación y hacemos click en ***Generate Project***:



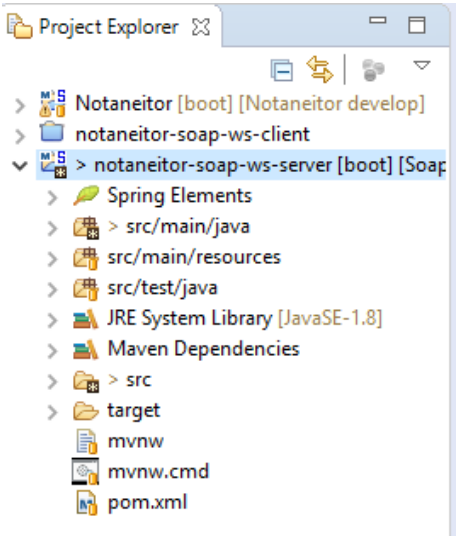
Al hacer click en Generate Project se descargará un paquete comprimido en formato zip con la estructura del proyecto, el cual debemos descomprimir y copiar la carpeta en nuestro workspace de STS.

Ahora debemos importar el proyecto a nuestro entorno de desarrollo de STS, para esto abrimos STS y hacemos click en ***File -> Import,*** buscamos el proyecto y hacemos click en ***Finish***, como se muestra en las siguientes imágenes:


File Edit Source Navigate Search Project Run
New Alt+Shift+N
Open File...
Open Projects from File System...
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Switch Workspace
Restart
Import...
Export...
Properties Alt+Enter
Exit
Import
Select
Create new projects from an archive file or directory.
Select an import wizard:
type filter text
General
Archive File
Existing Projects into Workspace
Preferences
Projects from Folder or Archive
< Back
Next >
Finish
Cancel


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
C:\Users\virtual_user\Source\Repositories\workspace-sts-3
Browse...
Select archive file:
Browse...
Projects:
notaneitor-soap-ws-server (C:\Users\virtual_user\Source\Repositories\workspac
Select All
Deselect All
Refresh
Options
Search for nested projects
Copy projects into workspace
Hide projects that already exist in the workspace
Working sets
Add project to working sets
New...
Working sets:
Select...
< Back
Next >
Finish
Cancel

Al finalizar la importación abremos creado el proyecto, como se muestra en la siguiente imagen.



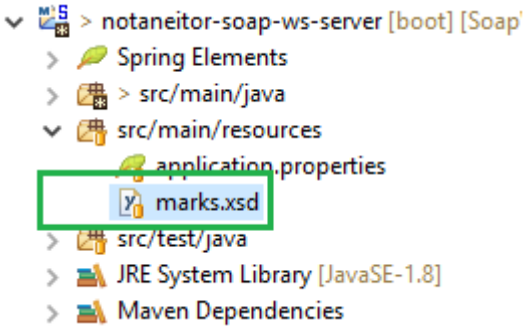
1.1.2 Añadir la dependencia Spring-WS

Cuando creamos un servicio web en SOAP los datos se exponen en formato ***WSDL (Web Services Description Language***) que contiene las normas para definir los mensajes, la ubicación del servicio web, los enlaces y las operaciones disponibles. Para poder generar la información en este formato es necesario añadir las dependencias de maven ***spring-boot-starter-web-services y wsdl4j*** al fichero pom.xml de la aplicación.

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-web-services</artifactId>
</dependency>

<dependency>
	<groupId>wsdl4j</groupId>
	<artifactId>wsdl4j</artifactId>
</dependency
```

1.1.3 Crear un XML schema para definir el dominio

Normalmente, el dominio del servicio web se define en un archivo ***de XML schema (XSD)*** que Spring exportará automáticante como WSDL y que será lo que se consuma por un cliente. En la carpeta ***src/main/resources*** de nuestro proyecto creamos el fichero ***mark.xsd***. Este fichero describe la estructura en formato XML de los elementos, atributos y operaciones que ofrecerá el servicio web SOAP.



Copiamo el siguiente contenido en el fichero. Este fichero tendrá los siguientes elementos definidos:

- **getMarksRequest**: Método que permitirá hacer una petición al servicio web y que enviará como parámetro el ***dni*** del alumno.
- **getMarksResponse**: Método que devolverá un elemento de tipo ***user*** .
- **User**: Elemento que representa un objeto complejo que contiene el ***dni, nombre y notas (mark)*** de un usuario (alumno).
- ***Mark***: Elemento que representa un objeto complejo que contiene el ***descripción y puntuación(score)*** de una nota (podemos llamarle asignatura).

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://uniovi.com/soap/ws"
           targetNamespace="http://uniovi.com/soap/ws"  elementFormDefault="qualified">

    <xs:element name="getMarksRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="dni" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="getMarksResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="user" type="tns:user"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="user">
        <xs:sequence>
            <xs:element name="dni" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="mark" type="tns:mark" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="mark">
        <xs:sequence>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="score" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

1.1.4 Generar clases de dominio basadas en un esquema XML

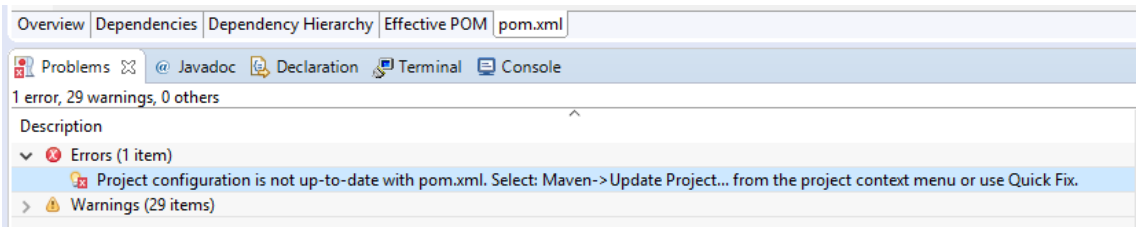
El siguiente paso es ***generar clases Java*** desde el archivo XSD que nos permitar gestionar de manera mas fácil el servicio Web. El enfoque correcto es hacer esto automáticamente durante el tiempo de compilación usando el plugin de maven ***jaxb2-maven-plugin***. Para incluir este plugin en nuestro proyecto vamos al fichero ***POM.XML*** y lo incluimos. Este plugin usa la herramienta XJC como motor de generación de código. ***XJC compila un archivo XML schema y genra las clases Java totalmente anotadas apartir de este de fichero xsd.***

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>jaxb2-maven-plugin</artifactId>
            <version>1.6</version>
            <executions>
                <execution>
                    <id>xjc</id>
                    <goals>
                        <goal>xjc</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <schemaDirectory>${project.basedir}/src/main/resources/</schemaDirectory>
                <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
                <clearOutputDir>false</clearOutputDir>
            </configuration>
        </plugin>
    </plugins>
</build>
```

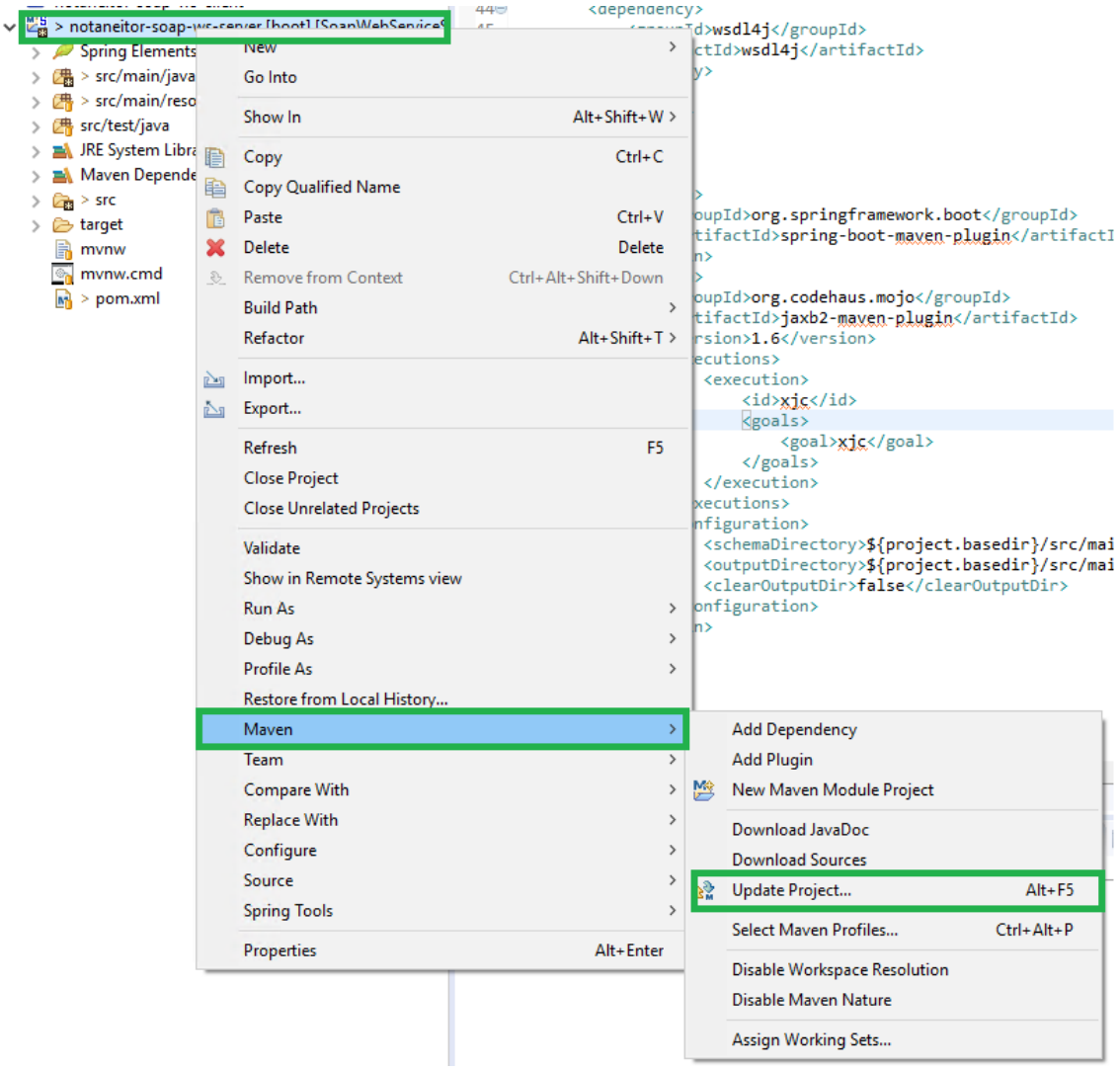
En este fichero es importante destacar dos elementos importantes:

- **schemaDirectory**: Especifica la carpeta del proyecto donde se almacena el fichero XML schema que se utilizará para generar las clases JAVA.
- **outputDirectory:** Especifica la carpeta del proyecto donde se generarán automáticamente las clases JAVA a partir de la definición del fichero XML schema.

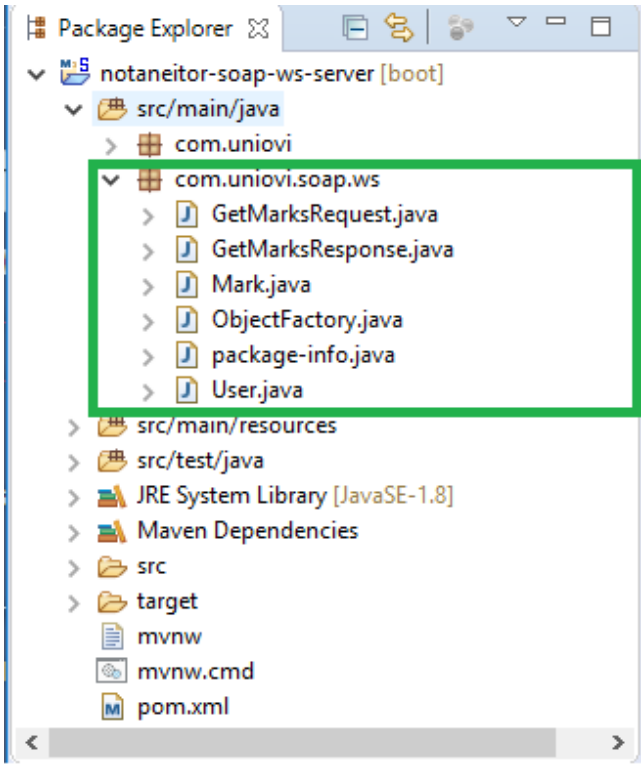
Al incluir el plugin es posible que genere un error de compilación como se muestra a continuación, debido a que no se ha actualizado el proyecto maven.



Para solucionar el error actualizamos el proyecto haciendo doble click sobre el mismo y luego ir al opción ***Maven-> Update Project***, como se muestra en la siguiente imagen:



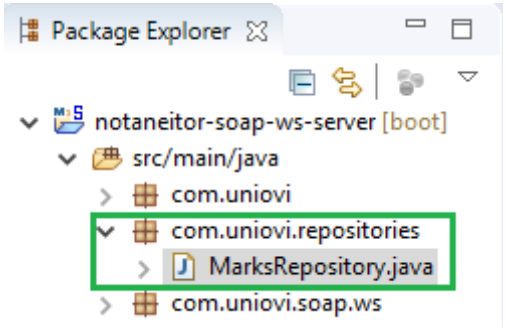
Al actualizar el proyecto se puede ver que se ha creado un paquete ***com.uniovi.soap.ws*** y dentro de este todas las clases JAVA necesarias para comezar a exponer el servicio web SOAP. Se puede observar que por cada elemento del fichero XML Schema se ha generado la clase correspondiente.



1.1.5 Crear un repositorio de notas

Con el objetivo de proporcionar datos al servicio web es necesario crear un repositorio de datos. En nuestro caso, vamos a crear un repositorio de notas.

Creamos un paquete ***com.uniovi.repositories*** y dentro de esta una clase ***MarksRepository.***



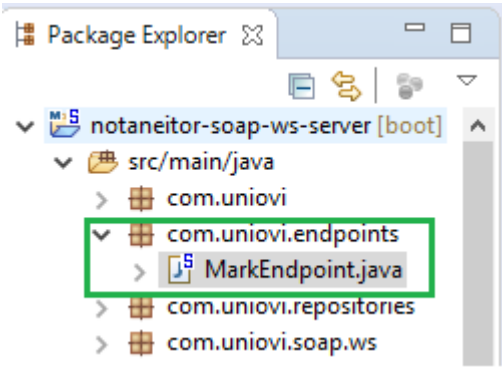

```
package com.uniovi.repositories;

import javax.annotation.PostConstruct;
import org.springframework.stereotype.Component;
import com.uniovi.soap.ws.Mark;
import com.uniovi.soap.ws.User;
import java.util.HashMap;
import java.util.Map;
```

```
import org.springframework.util.Assert;

@Component
public class MarksRepository {
	private static final Map<String, User> marks = new HashMap<>();

	@PostConstruct
	public void initData() {
		User student = new User();
		student.setName("Jose");
		student.setDni("75999999X");

		Mark mark1 = new Mark();
		mark1.setDescription("SDI");
		mark1.setScore(10);

		Mark mark2 = new Mark();
		mark2.setDescription("DLP");
		mark2.setScore(8);

		Mark mark3 = new Mark();
		mark3.setDescription("IP");
		mark3.setScore(8);

		student.getMark().add(mark1);
		student.getMark().add(mark2);
		student.getMark().add(mark3);

		marks.put(student.getDni(), student);
	}

	public User findAllByUser(String dni) {
		Assert.notNull(dni, "The user's DNI must not be null");
		return marks.get(dni);
	}
}
```

Nota: de momento son datos que no estarán almacenado en ningúna base de datos, si no que lo pondremos Harcodeado en nuestro proyecto.

1.1.6 Crear un Endpoint de notas

Para manejar las solicitudes SOAP entrantes realizadas por los clientes es necesario crear un EndPoint (punto de entrada) a nuestro servicio web. Para esto, Creamos un paquete ***com.uniovi.endpoints*** y dentro de esta una clase ***MarkEndpoint.***



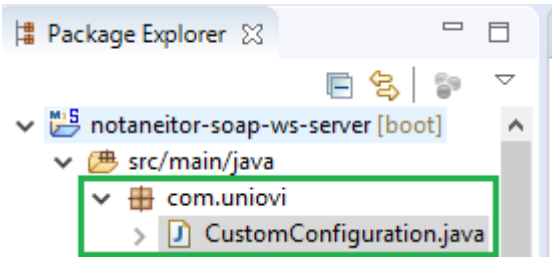

```java
package com.uniovi.endpoints;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
import org.springframework.ws.server.endpoint.annotation.RequestPayload;
import org.springframework.ws.server.endpoint.annotation.ResponsePayload;
import com.uniovi.repositories.MarksRepository;
import com.uniovi.soap.ws.GetMarksRequest;
import com.uniovi.soap.ws.GetMarksResponse;

@Endpoint
public class MarkEndpoint {
	private static final String NAMESPACE_URI = "http://uniovi.com/soap/ws";
	private MarksRepository markRepository;

	@Autowired
	public MarkEndpoint(MarksRepository markRepository) {
		this.markRepository = markRepository;
	}

	@PayloadRoot(namespace = NAMESPACE_URI, localPart = "getMarksRequest")
	@ResponsePayload
	public GetMarksResponse getMarks(@RequestPayload GetMarksRequest request) {
		GetMarksResponse response = new GetMarksResponse();
		response.setUser(markRepository.findAllByUser(request.getDni()));
		return response;
	}
}
```

En esta clase se utilizan una serie de anotaciones las cuales son importantes destacar:

- **@Endpoint:** registra la clase con Spring WS como candidato potencial para procesar los mensajes SOAP entrantes al servicio web.
- **@PayloadRoot:** es luego utilizado por Spring WS para elegir el método del controlador a ejecutar, basando en el espacio de nombres del mensaje y localPart definido en el servicio.
- **@RequestPayload:** indica que el mensaje entrante será mapeado al parámetro de solicitud del método.
- **@ResponsePayload:** hace que Spring WS asigne el valor devuelto a la respuesta Payload (ResponsePayload)

1.1.7 Configurar los beans del servicio web

Creamos una nueva clase ***CustomConfiguration*** que herede de ***WsConfigurerAdapter***, en esta clase se especifica la configuración de los beans relacionados con los Web Services de Spring.




```
package com.uniovi;

import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.ws.config.annotation.EnableWs;
import org.springframework.ws.config.annotation.WsConfigurerAdapter;
import org.springframework.ws.transport.http.MessageDispatcherServlet;
import org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition;
import org.springframework.xml.xsd.SimpleXsdSchema;
import org.springframework.xml.xsd.XsdSchema;

@EnableWs
@Configuration
public class CustomConfiguration extends WsConfigurerAdapter {

	@Bean
	public ServletRegistrationBean messageDispatcherServlet(ApplicationContext
applicationContext) {
		MessageDispatcherServlet servlet = new MessageDispatcherServlet();
		servlet.setApplicationContext(applicationContext);
		servlet.setTransformWsdlLocations(true);
		return new ServletRegistrationBean(servlet, "/webservice/*");
	}

	@Bean(name = "marks")
	public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema marksSchema) {
		DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();
		wsdl11Definition.setPortTypeName("MarksPort");
		wsdl11Definition.setLocationUri("/webservice/marks");
		wsdl11Definition.setTargetNamespace("http://uniovi.com/soap/ws");
		wsdl11Definition.setSchema(marksSchema);
		return wsdl11Definition;
	}

	@Bean
	public XsdSchema marksSchema() {
		return new SimpleXsdSchema(new ClassPathResource("marks.xsd"));
	}
}
```

Spring WS utiliza ***MessageDispatcherServlet*** que es un tipo de servlet para manejar mensajes SOAP. Es importante inyectar y establecer ApplicationContext a MessageDispatcherServlet para que Spring WS detecte el bean automáticamente.

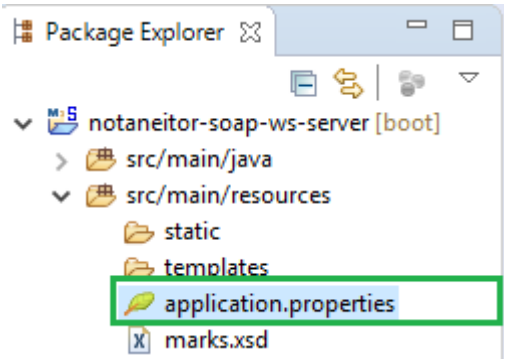
Es necesario especificar los nombres de los beans para MessageDispatcherServlet y DefaultWsdl11Definition. Estos nombres son los que determinan la URL bajo la cual el servicio web y el archivo WSDL generado están disponibles. En este caso, el fichero WSDL estará disponible en

http: //<host> :<Puerto> /webservice/marks.wsdl.

Ejemplo: ***http://localhost:8090 /webservice/marks.wsdl.***

1.1.8 Configurar el Puerto

Finalmente configuramos el puerto por el cual va escuchar el servicio web SOAP al desplegar la aplicación. Para esto modificamos el fichero ***application.properties*** de la aplicación.

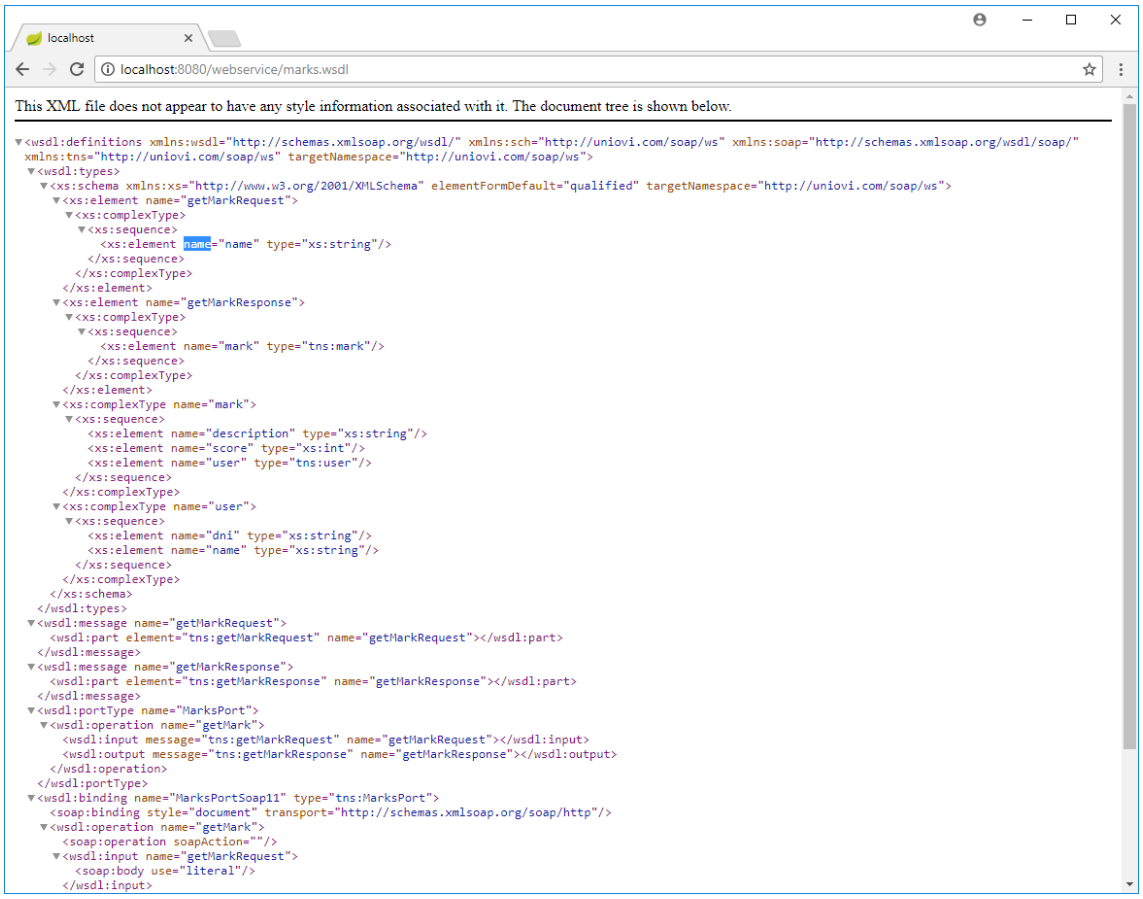



```
server.port=8090
```

1.1.9 Ejecutamos la aplicación y obteniendo el WSDL

Ahora ejecutamos la aplicación ***haciendo click derecho->Run As -> Java applicaction*** en el proyecto y finalmente probamos en un navegador con la siguiente url y veremos que el fichero WSDL está disponible.

http://localhost:8090/webservice/marks.wsdl



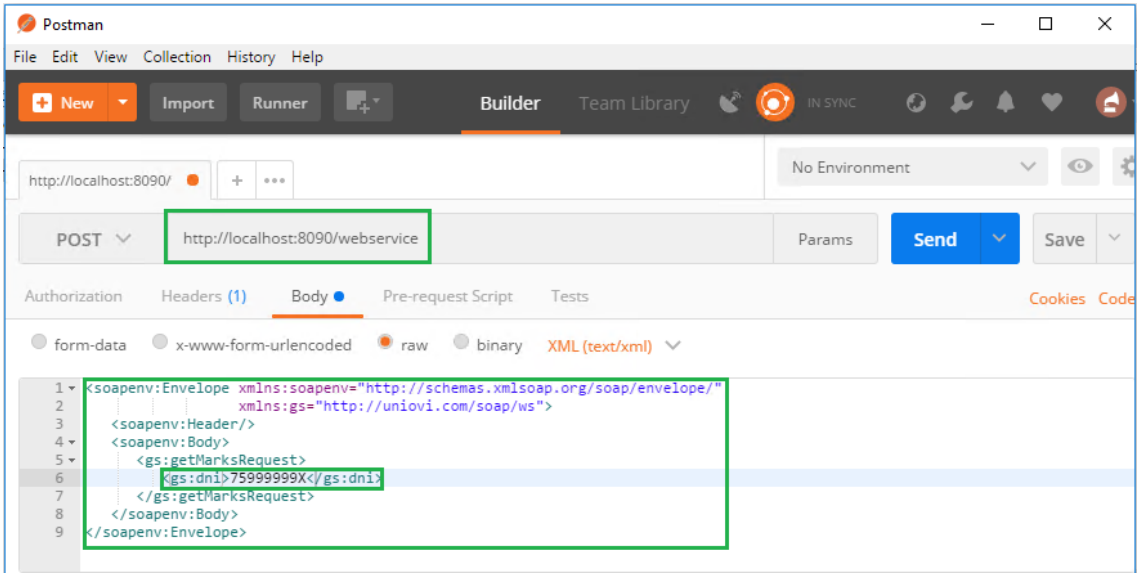
1.1.10 Probando el Web Service WSDL

Una vez ejecutada y desplegada la aplicación podemos probar el servicio web haciendo una petición SOAP al servicio para que nos devuelva en nuestro caso las notas de un alumno especifico. Para esto, podemos utilizar una herramienta para probar servicios Rest y SOAP como postman o SoapUI.

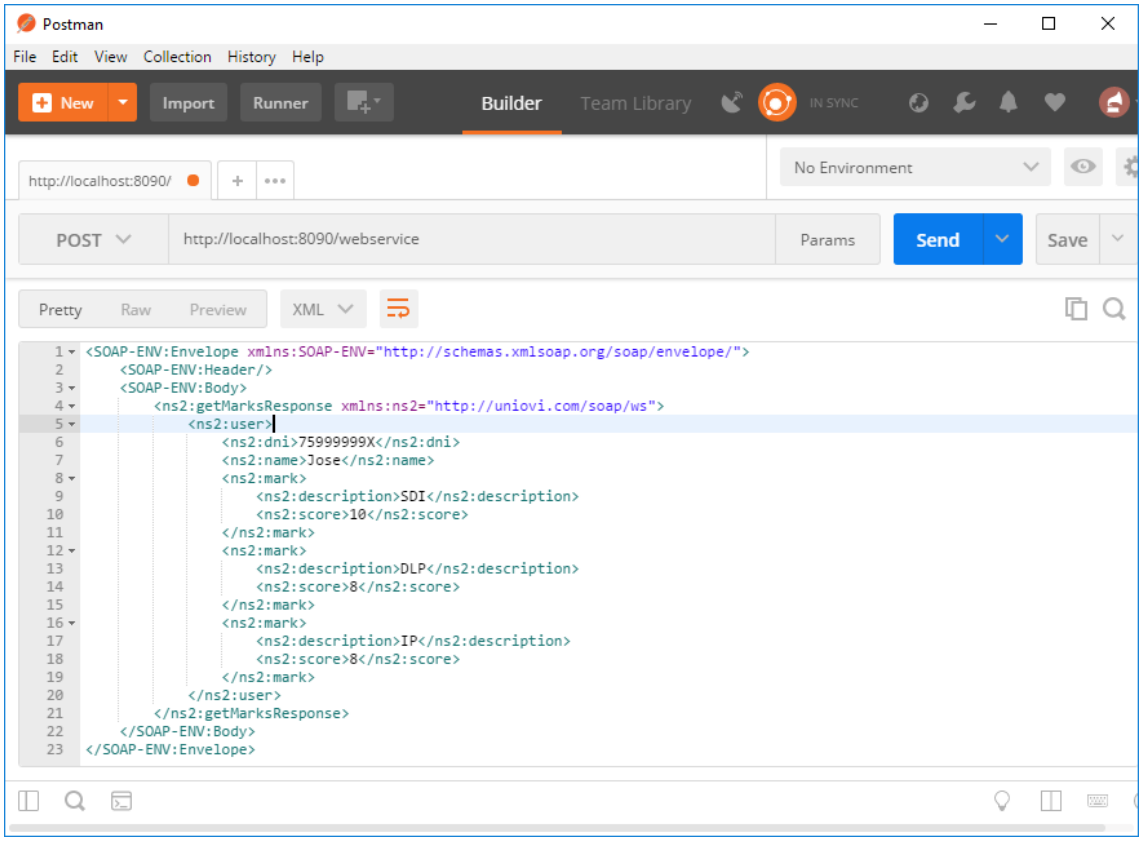
En nuestro caso utilizaremos Postman, solo tenemos que hacer una petición POST y configurar el ***HEADER*** como ***"content-type: text/xml"*** y en el cuerpo del mensaje pasarle un sobre(***soapenv:Envelope)*** con los parámetros necesarios, en este caso estamos pidiendo las notas para un alumno cuyo dni es ***75999999X***.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:gs="http://uniovi.com/soap/ws">
   <soapenv:Header/>
   <soapenv:Body>
      <gs:getMarksRequest>
         <gs:dni>75999999X</gs:dni>
      </gs:getMarksRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Peticion en PostMan



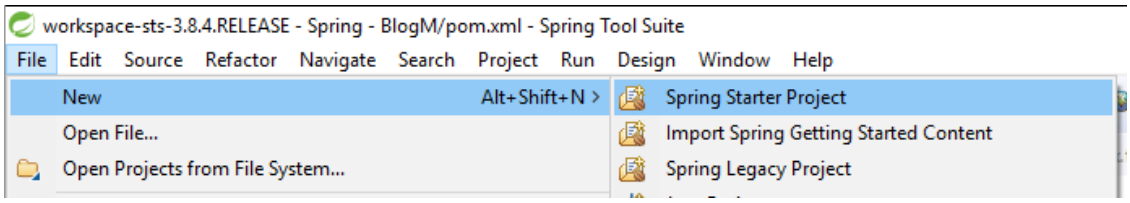
Respuesta



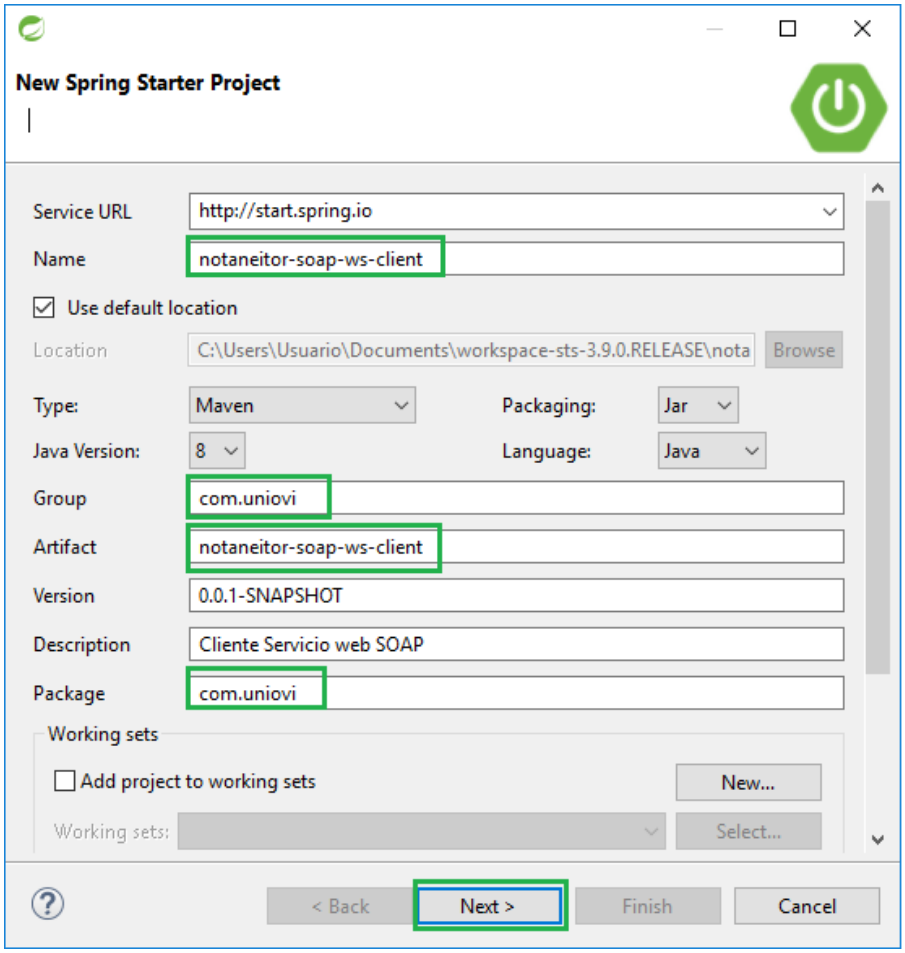
1.2 Consumiendo el servicio Web

En esta parte construiremos una aplicación web cliente que obtenga datos publicados de un servicio web SOAP en lugar un repositorio de datos interno, como, por ejemplo, una base de datos. En nuestro caso, consumiremos los datos del servicio web de notas desarrollado anteriormente.

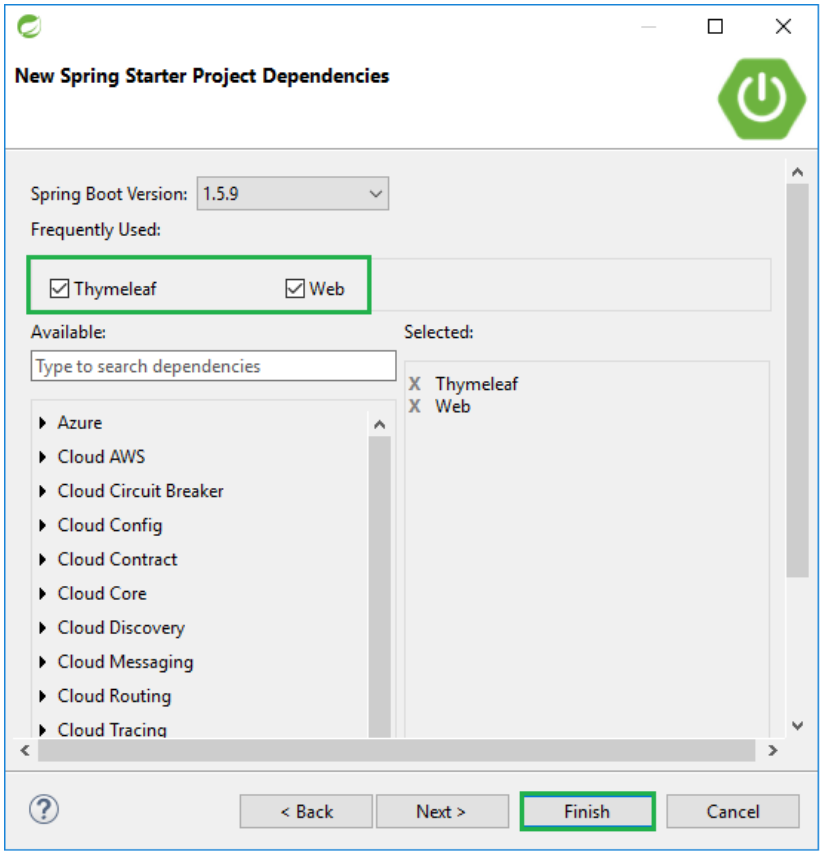
Lo primero es crear un proyecto "Spring Starter Project" como se muestra a continuación: **File -> New -> Spring Starter Project**



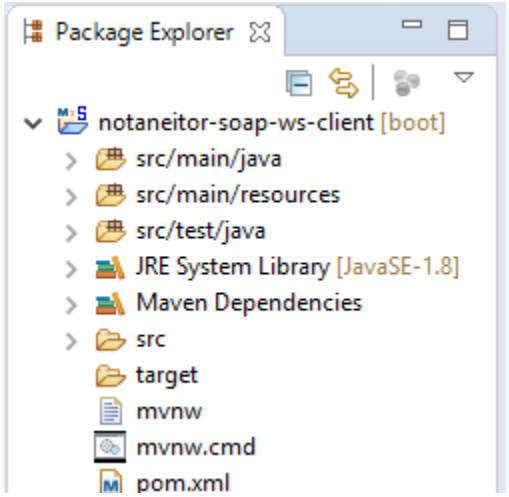
Le damos el nombre "*notaneitor-soap-ws-client*" , escribimos los datos del proyec como se muestra en la siguiente imagen y lu luego hacemos click en el botón ***Next***..



Seleccionamos las dependencias Web y Thymeleaf que necesitaremos mas adelante en el proyecto y hacemos click en el botón ***Finish***



Con esto ya tendremos el proyecto creado como se muestra la siguiente imagen:



1.2.1 Añadir la dependencia Spring-WS

Para poder poder trabajar con servicios web SOAP es necesario añadir las dependencias de maven ***spring-ws-core*** al fichero ***pom.xml*** de la aplicación

```
<dependency>
    <groupId>org.springframework.ws</groupId>
    <artifactId>spring-ws-core</artifactId>
</dependency>
```

1.2.2 Generar los objetos del dominio basado en WSDL

El siguiente paso es ***generar clases Java*** desde un WSDL que podemos obtener desde un servicion web SOAP. Similar al caso anterior, para generar las clases JAVA apartir de un WSDL utilizaremos el plugin ***maven-jaxb2-plugin***.

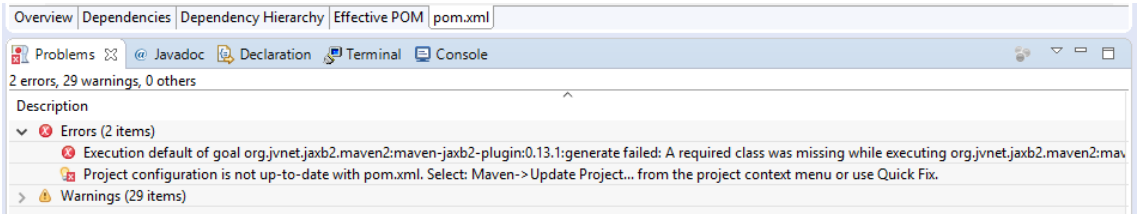
Para incluir este plugin en nuestro proyecto vamos al fichero ***POM.XML*** y lo incluimos.

```
<plugin>
    <groupId>org.jvnet.jaxb2.maven2</groupId>
    <artifactId>maven-jaxb2-plugin</artifactId>
    <version>0.13.1</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <schemaLanguage>WSDL</schemaLanguage>
        <generatePackage>com.uniovi.wsdl</generatePackage>
        <generateDirectory>${project.basedir}/src/main/java</generateDirectory>
        <clearOutputDir>false</clearOutputDir>
        <schemas>
            <schema>
                <url>http://localhost:8090/webservice/marks.wsdl</url>
            </schema>
        </schemas>
    </configuration>
</plugin>
```

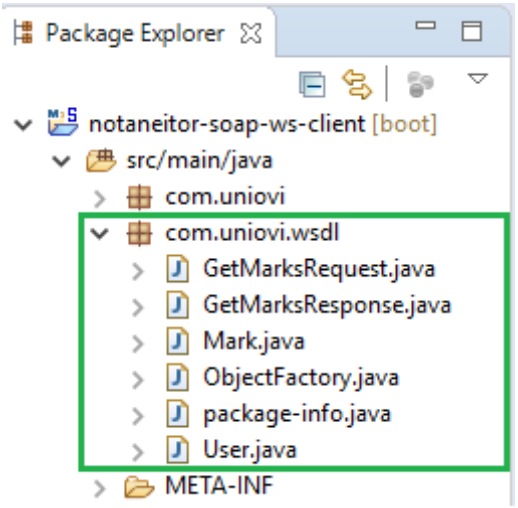
En este fichero es importante destacar dos elementos importantes:

- **generatePackage:** Especifica el nombre del paquete donde se generarán automáticamente las clases JAVA a partir de la definición del fichero WSDL.
- **generateDirectory:** Especifica la carpeta del proyecto donde se generarán automáticamente las clases JAVA a partir de la definición del fichero WSDL.
- **Schemas:** dentro de este nodo se especifica las diferentes localizaciones (URL) de los ficheros WSDL que se exponen atravéz de servicios web SOAP.

Nota: Al incluir el plugin es posible que genere un error de compilación como se muestra a continuación, debido a que el plugin está intentando acceder al fichero WSDL localizado en la URL `http://localhost:8080/webservice/marks.wsdl`. Esto se debe a que el servicio web SOAP no está disponible. Para solucionarlo solo tenemos que desplegar la apliacción del servicio web SOAP definido en el proyecto anterior. Solo tenemos que ejecutar la aplicación ***haciendo click derecho->Run As -> Java applicaction*** en el proyecto ***(notaneitor-soap-ws-server).***



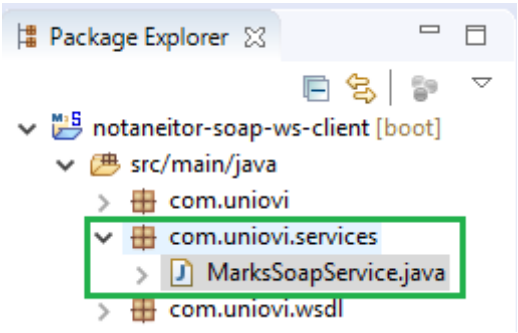
Luego para solucionar el error actualizamos el proyecto ***(notaneitor-soap-ws-client).*** haciendo doble click sobre el mismo y luego ir al opción ***Maven-> Update Project.*** Esto corregiraá los errores y generará las clases JAVA corespondientes, como se muestra en la siguiente imagen:



1.2.3 Crear un servicio web cliente

Crear un servicio web cliente para obtener las notas de los alumnos llamando el servicio web SOAP(server) publicado anteriomente.

Para crear un servicio web cliente, simplemente hay que crear una clase que herede de la clase ***WebServiceGatewaySupport*** y definir sus operaciones. Para hacer esto vamos a crear el paquete ***com.univi.services*** y dentro la clase ***MarksSoapService***.

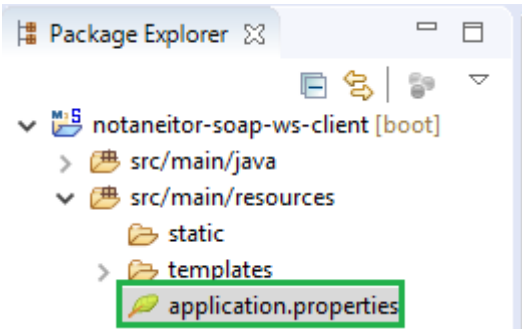



```java
package com.uniovi.services;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.ws.client.core.support.WebServiceGatewaySupport;
import org.springframework.ws.soap.client.core.SoapActionCallback;
import com.uniovi.wsdl.GetMarksRequest;
import com.uniovi.wsdl.GetMarksResponse;

public class MarksSoapService extends WebServiceGatewaySupport {

    @Value("${service.endpoint}")
    private String serviceEndpoint;

    @Value("${service.soap.action}")
    private String serviceSoapAction;

    public GetMarksResponse getMarks(String dni) {
        GetMarksRequest request = new GetMarksRequest();
        request.setDni(dni);
        GetMarksResponse response = (GetMarksResponse)
getWebServiceTemplate().marshalSendAndReceive(serviceEndpoint,
                request, new SoapActionCallback(serviceSoapAction));

        return response;
    }
}
```

Este serviciote contiene el método ***getMarks()*** que es el que hace el intercambio real con el servicio web SOAP. Para enviar y recibir mensajes del SW SOAP se utilizan las clases ***GetMarksRequest y GetMarksResponse*** que fueron generadas en el proceso de generacion de Código apartir del WSDL utilizando ***JAXB*** (Descrito en el paso anterior).

1.2.4 Modificar el fichero applications.properties

En el fichero de configuración definimos el ***puerto*** por el que escuchará la aplicación web cliente, así como las variables del ***endpoint y el método*** que se injectará en diferentes puntos de la aplicación y que se utilizan para invocar llamar el servicio web SOAP disponible en estas urls.

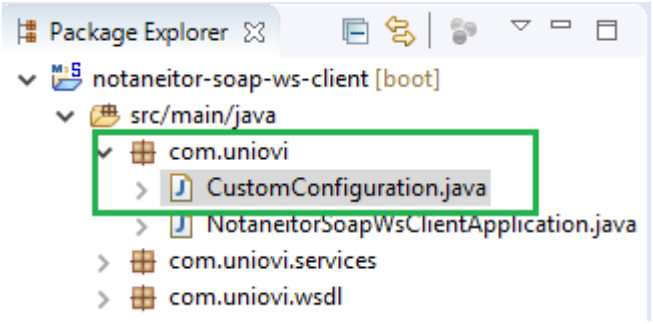



```
server.port=8091
service.endpoint=http://localhost:8090/webservice
service.soap.action=http://localhost:8090/webservice/GetMark
```

1.2.5 Configurar los componentes servicio web

Cuando se hace petiones a un servicio web SOAP la respuesta es devuelta en formato XML por lo cual es necesarios serializar y deserializar las petiones que se realizan. Spring WS utiliza el módulo ***OXM*** de Spring Framework que tiene el ***Jaxb2Marshaller*** para serializar y deserializar solicitudes XML

Para poder realizar esto creamos una nueva clase ***CustomConfiguration*** que incluya los beans necesarios para serializar las peticiones.




```java
package com.uniovi;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.oxm.jaxb.Jaxb2Marshaller;
import com.uniovi.services.*;

@Configuration
public class CustomConfiguration {
    @Value("${service.endpoint}")
    private String serviceEndpoint;

    @Bean
    public Jaxb2Marshaller marshaller() {
        Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        marshaller.setContextPath("com.uniovi.wsdl");
        return marshaller;
    }

    @Bean
    public MarksSoapService marksService(Jaxb2Marshaller marshaller) {
        MarksSoapService client = new MarksSoapService();
        client.setDefaultUri(serviceEndpoint);
        client.setMarshaller(marshaller);
        client.setUnmarshaller(marshaller);
        return client;
    }
}
```

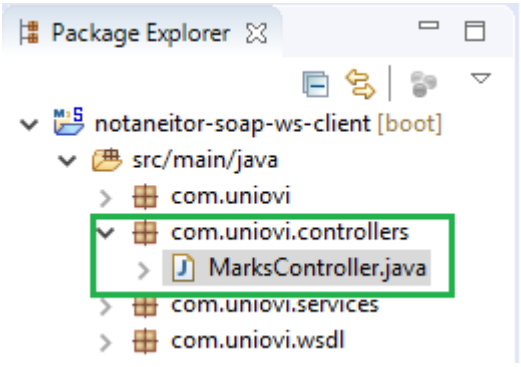
El ***marshaller*** apunta a la colección de objetos de dominio generados y los utilizará para serializar y deserializar entre ***XML y POJOs***.

El ***marksService*** se crea y configura con la URI del servicio de notas definida anteriormente. También está configurado para usar el Jaxb2Marshaller.

1.2.6 Crear controlador MarksController

Vamos a crear un controlador que se encargue de recibir las peticiones del cliente, llamar al servicio web ***MarksSoapService*** creado previamente y generar una respuesta.

Para hacer esto vamos a crear el paquete ***com.univi.controllers*** y dentro de este la clase ***MarksController***. . Este controlador contiene el método ***getMarks***() que recibirá como parámetro por Get el ***DNI*** de un usuario y devolverá una lista de notas de dicho usuario.

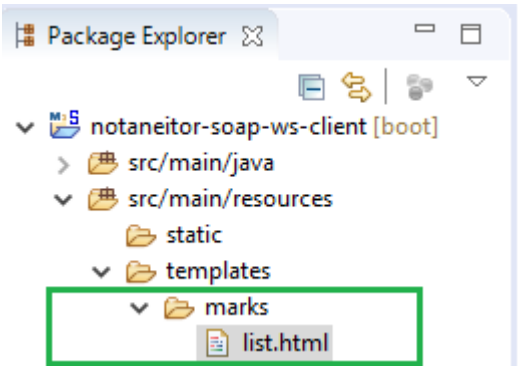



```
package com.uniovi.controllers;

import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import com.uniovi.services.*;
import com.uniovi.wsdl.*;

@Controller
public class MarksController {

    @Autowired
    private MarksSoapService marksSoapService;

    @RequestMapping("/marks/list")
    public String getMarks(Model model, @RequestParam String dni) {
        List<Mark> marks = new ArrayList<Mark>();
        User user = marksSoapService.getMarks(dni).getUser();
        if (user != null) {
            marks = user.getMark();
        }
        model.addAttribute("dni", dni);
        model.addAttribute("markList", marks);
        return "marks/list";
    }
}
```

1.2.7 Crear vista

Ahora vamos a crear la vista para mostrar los datos que nos devueve el controlador en formato HTML. En este caso se mostrará la lista de notas de un usuario especifico.

Para hacer esto, en la carpeta ***resource/template*** creamos una carpeta ***marks*** y dentro de esta un fichero ***list.html.***



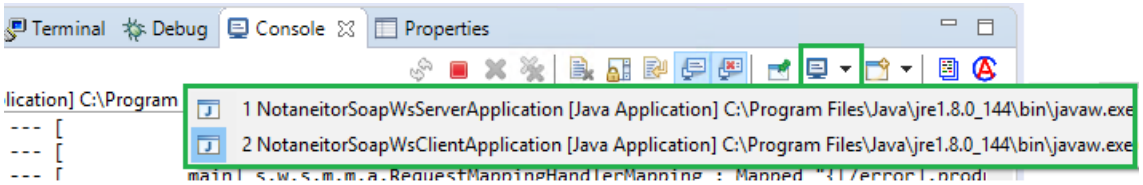

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Notaneitor - cliente SOAP</title>
<meta charset="utf-8" />
</head>
<body>
    <div class="container">
        <h2>
            Listado de notas de: <span th:text="${dni}">99999999K</span>
        </h2>
        <table>
            <thead>
                <tr>
                    <th>Descripción</th>
                    <th>Puntuación</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="mark : ${markList}">
                    <td th:text="${mark.description}">asignatura</td>
                    <td th:text="${mark.score}">10</td>
                </tr>
            </tbody>
        </table>
        <div th:if="${#lists.isEmpty(markList)}">Lista de notas vacía</div>
    </div>
</body>
</html>
```

1.2.8 Probando el servicio web cliente

Ahora ejecutamos la aplicación cliente(***notaneitor-soap-ws-client)*** ***haciendo click derecho->Run As -> Java Applicaction*** en el proyecto y finalmente probamos en un navegador con la siguiente url: **http://localhost:8091/marks/list?dni=75999999X**

Nota: Es necesario que el servicio web servidor este disponible, por lo que debemos asegurarnos de que la aplicación ***notaneitor-soap-ws-server*** se esté ejecutando.

Para comprobar que se están ejecutando las dos aplicaciones, vamos a la console de STS y visualizamos el despliegue como se muestra en la siguiente imagen:



Si hacemos una petición con un dni válido veremos que el servicio nos devuelve la lista de notas para ese usuario en concreto, por el contrario, y usamos un dni inválido (no existe en el servidor) el servicio nos devolverá una lista vacía.