



# Sistemas Distribuidos e Internet

## Curso 2023/2024

### Servicios Web SOAP con Spring Boot

#### Sesión 12 y 13



## Contenido

1	Servicios Web SOAP.....	3
1.1	Desarrollando el servicio Web SOAP.....	3
1.1.1	Creación del proyecto.....	3
1.1.2	Añadir la dependencia Spring-WS.....	7
1.1.3	Crear un XML schema para definir el dominio.....	7
1.1.4	Generar clases de dominio basadas en un esquema XML.....	9
1.1.5	Crear un repositorio de notas.....	11
1.1.6	Crear un Endpoint de notas.....	13
1.1.7	Configurar los beans del servicio web.....	14
1.1.8	Configurar el Puerto.....	16
1.1.9	Ejecutamos la aplicación y obteniendo el WSDL.....	17
1.1.10	Probando el Web Service WSDL.....	17
1.2	Consumiendo un servicio web.....	21
1.2.1	Añadir la dependencia Spring-WS.....	22
1.2.2	Generar los objetos del dominio basado en WSDL.....	23
1.2.3	Crear un servicio web cliente.....	25
1.2.4	Modificar el fichero application.properties.....	26
1.2.5	Configurar los componentes servicio web.....	26
1.2.6	Crear controlador MarksController.....	28
1.2.7	Crear vista.....	29
1.2.8	Probando el servicio web cliente.....	30
1.3	Resultado esperado en los repositorios de GitHub.....	33



# 1 Servicios Web SOAP

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

Crearemos un **nuevo repositorio PRIVADO en GitHub con el nombre sdix-lab-soap-sw** (sustituyendo la x por el IDGIT correspondiente).

#### !!!!MUY IMPORTANTE!!!!

Aunque en este guión se ha usado como nombre de repositorio para todo el ejercicio **sdix-lab-soap-sw**, cada alumno deberá usar como nombre de repositorio **sdix-lab-soap-sw**, donde **x** = columna **IDGIT** en el Documento de ListadoAlumnosSDI2324.pdf del CV.

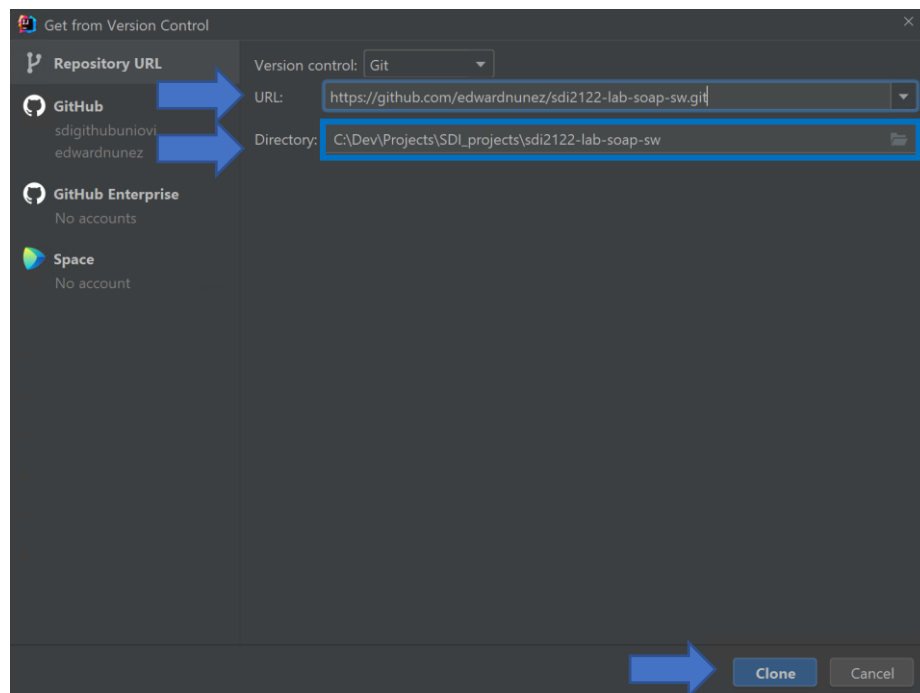
**Por ejemplo el alumno con IDGIT=2324-101, deberá crear un repositorio con nombre sdi2324-101-lab-soap-sw y un proyecto en el IDE con nombre también sdi2324-101-lab-soap-sw.**

Otra cuestión **IMPRESINDIBLE** es que una vez creado el repositorio deberá invitarse como colaborador a dicho repositorio a la cuenta github denominada **sdigithubuniovi**.

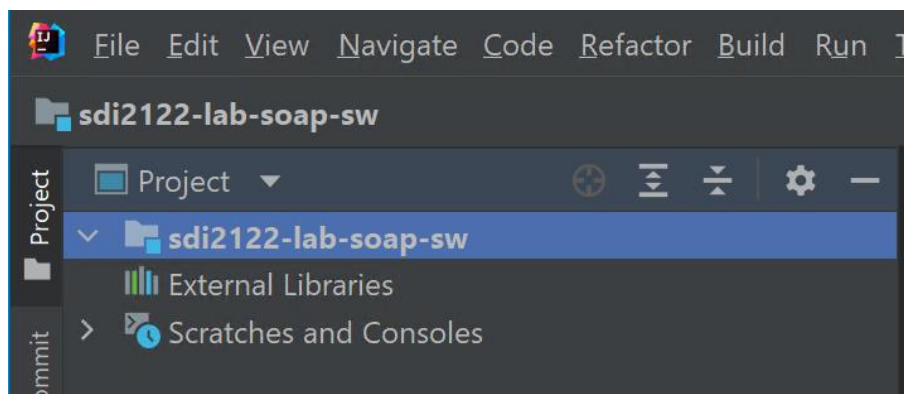
**En resumen:**

- Nombre repositorios en GitHub: **sdi2324-101-lab-soap-sw**
- Nombre repositorios en local **sdi2324-101-lab-soap-sw**
- Colaborador invitado: **sdigithubuniovi**

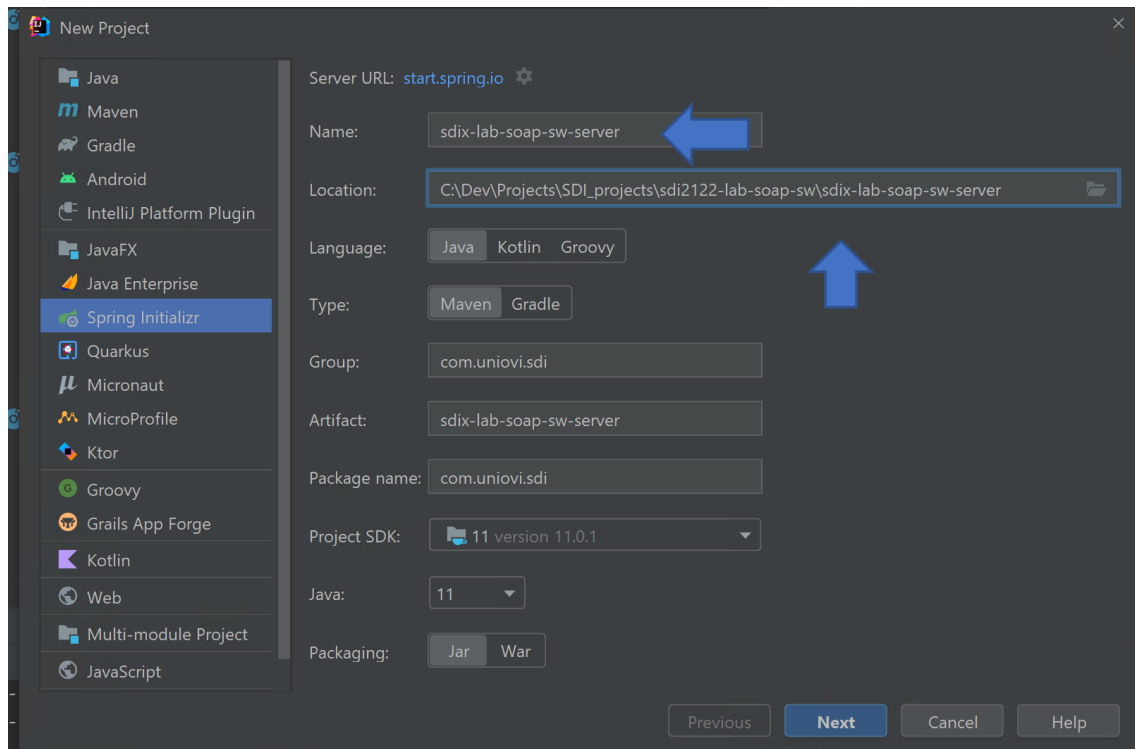
En el IntelliJ IDEA importaremos el proyecto a través de **GIT**. Vamos al menú **File/New/Project from Version Control** y copiamos la URL del repositorio de GitHub para importarlo en el IDE. Además, en **Directory** podemos especificar la carpeta local en la que se almacenará el proyecto. Por último, hacemos clic en el botón **Clone** para clonar el repositorio.



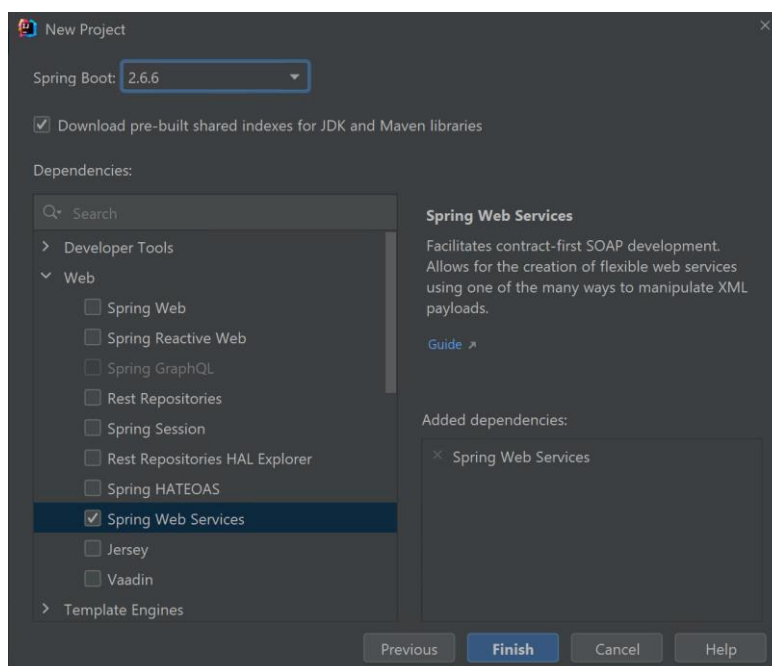
Una vez clonado el repositorio aparecerá el proyecto en el IDE, como se muestra en la siguiente imagen:



A continuación, vamos a crear un nuevo **proyecto Springboot** que llamaremos “**sdix-lab-soap-sw-server**”. Para esto vamos al menú **File | New | Project | Spring Initializr**, completamos la información, y lo guardamos en la carpeta del repositorio local, como se muestra en la siguiente pantalla:

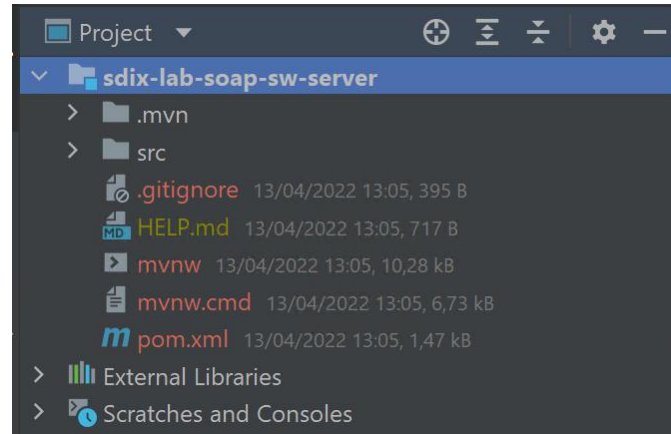


A continuación, en la siguiente pantalla **indicamos que utilizaremos la dependencia de Spring Web Services** y pulsamos el botón **Finish** para finalizar el proceso:

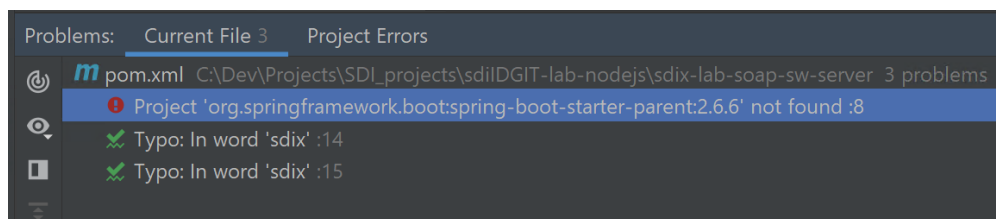




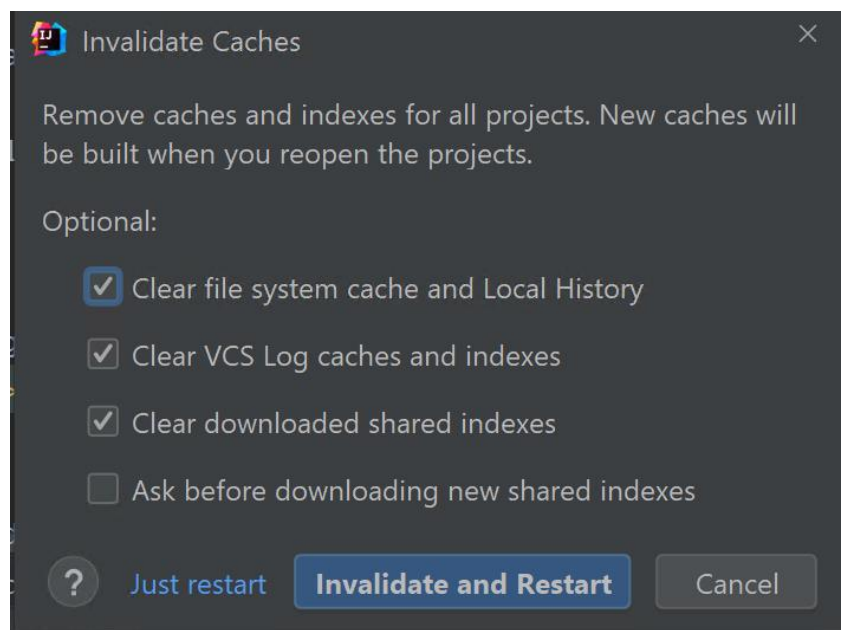
Al finalizar la importación abremos creado el proyecto, como se muestra en la siguiente imagen:



Si al crear el proyecto ocurre el error que se muestra en la siguiente imagen:



Para solucionar el error anterior, desde el menú del IDE **File | Invalidate Caches**, seleccionamos las opciones que se muestran en la siguiente imagen, y pulsamos el botón **Invalidate and Restart**:



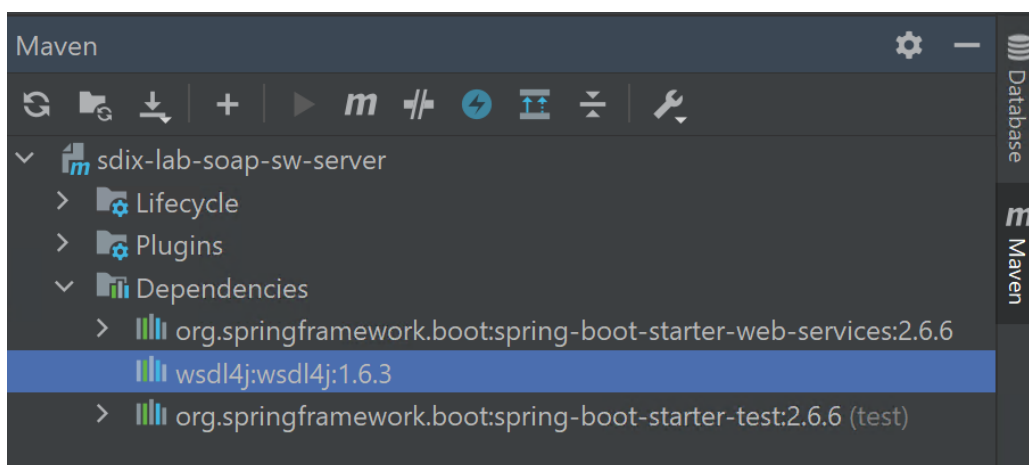


### 1.1.2 Añadir la dependencia Spring-WS

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

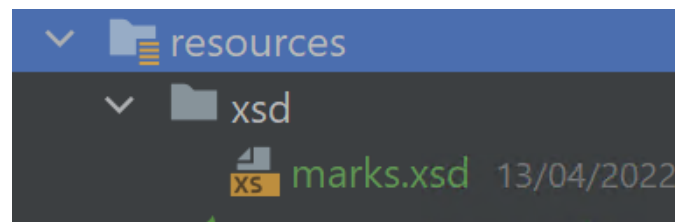
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web-services</artifactId>
</dependency>
<dependency>
  <groupId>wsdl4j</groupId>
  <artifactId>wsdl4j</artifactId>
</dependency>
```

A continuación, vamos al panel de Maven y recargamos las dependencias:



### 1.1.3 Crear un XML schema para definir el dominio

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



Copiamos el siguiente contenido en el fichero, que 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 alumno (user).
- **Mark**: Elemento que representa un objeto complejo que contiene la **descripción** y **puntuación(score)** de una nota (podemos llamarle asignatura).

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://uniovi.com/sdi/soap/ws"
  targetNamespace="http://uniovi.com/sdi/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 permitan gestionar cómodamente el servicio web. El enfoque correcto es hacer esto **automáticamente en tiempo de compilación**, usando el plugin (añadiéndolo en el **pom.xml**) de maven **jaxb2-maven-plugin**. Este plugin usa la herramienta XJC como motor de generación de código. **XJC genera clases Java anotadas a partir de un 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>2.5.0</version>
      <executions>
        <execution>
          <id>xjc</id>
          <goals>
            <goal>xjc</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
```

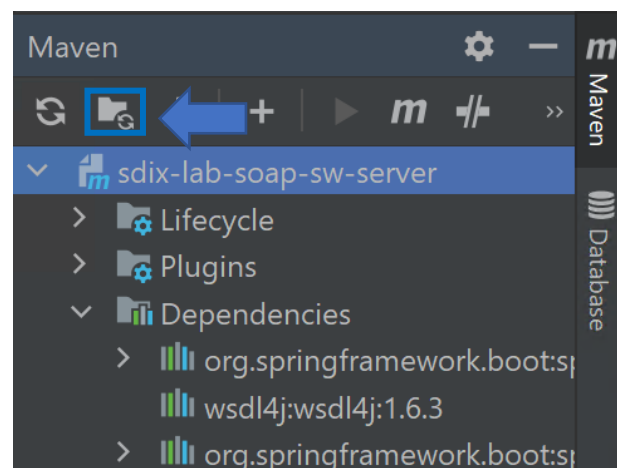


```
<sources>
  <source>src/main/resources/xsd</source>
</sources>
<outputDirectory>src/main/java</outputDirectory>
<clearOutputDir>false</clearOutputDir>
</configuration>
</plugin>
</plugins>
</build>
```

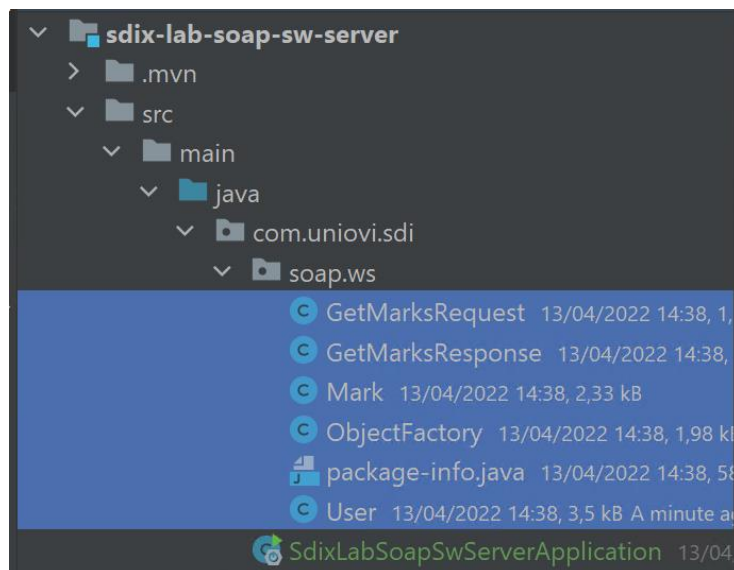
Los elementos añadidos al POM.XML, sirven para:

- **source:** 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 continuación, actualizamos todos los recursos del proyecto. Vamos al panel de **Maven** y pulsamos el botón **Actualizar all projects**:



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



Nota: En función de la versión de java y de spring que estemos utilizando hay que cambiar los import de los paquetes de **javax.xx** (spring 2.6.3) por **jakarta.xx** (spring 3.2.X).

Por ejemplo, cambiar:

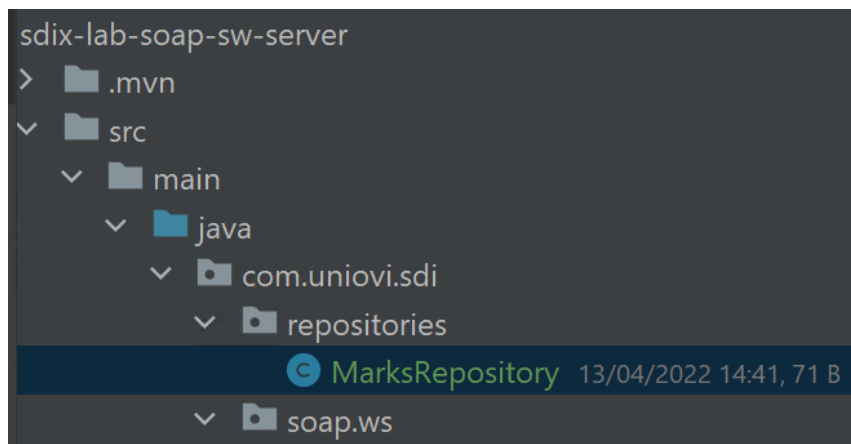
```
import javax.xml.bind.annotation.XmlAccessorType;
```

por

```
import jakarta.xml.bind.annotation.XmlAccessorType;
```

### 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 que proporcionará los datos directamente, en lugar de obtenerlos de la una base de datos**. Creamos un paquete **com.uniovi.sdi.repositories** y le añadimos una clase **MarksRepository**.



*El contenido de la clase MarksRepository será el siguiente:*

```
package com.uniovi.sdi.repositories;

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

@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);

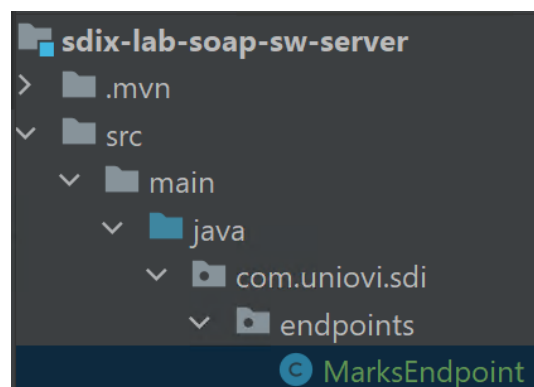
student.getMark().add(mark1);
student.getMark().add(mark2);
student.getMark().add(mark3);

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);
}
}
```

### 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. Así pues, creamos un paquete *com.uniovi.sdi.endpoints* y dentro de esta una clase *MarksEndpoint*.



*El contenido de la clase MarksEndpoint será el siguiente:*

```
package com.uniovi.sdi.endpoints;

import com.uniovi.sdi.repositories.MarksRepository;
import com.uniovi.sdi.soap.ws.GetMarksRequest;
import com.uniovi.sdi.soap.ws.GetMarksResponse;
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;

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

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

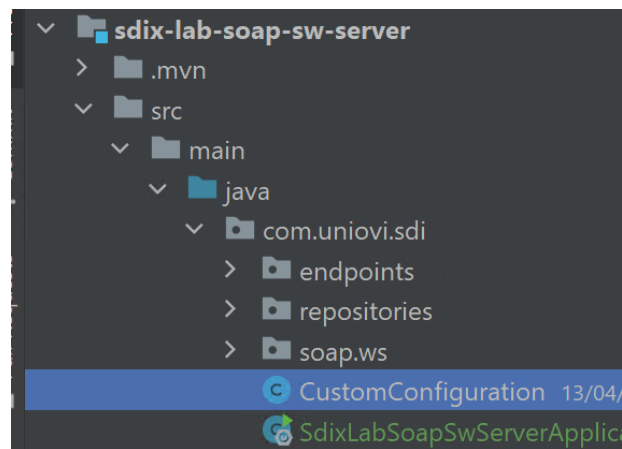
    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getMarksRequest")
    @ResponsePayload
    public GetMarksResponse getMarks(@RequestPayload GetMarksRequest request) {
        GetMarksResponse response = new GetMarksResponse();
        response.setUser(marksRepository.findAllByUser(request.getDni()));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

A continuación, se explican las diferentes anotaciones incluidas en la clase:

- **@Endpoint:** registra la clase con Spring WS como candidato potencial para procesar los mensajes SOAP entrantes al servicio web.
- **@PayloadRoot:** utilizado por Spring WS para elegir el método del controlador a ejecutar, en base al 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.



*El contenido de la clase CustomConfiguration será el siguiente:*

```
package com.uniovi.sdi;

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 DefaultWsd11Definition defaultWsd11Definition(XsdSchema marksSchema)
{
    DefaultWsd11Definition wsdl11Definition = new DefaultWsd11Definition();
    wsdl11Definition.setPortTypeName("MarksPort");
    wsdl11Definition.setLocationUri("/webservice/marks");
    wsdl11Definition.setTargetNamespace("http://uniovi.com/sdi/soap/ws");
    wsdl11Definition.setSchema(marksSchema);
    return wsdl11Definition;
}

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

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

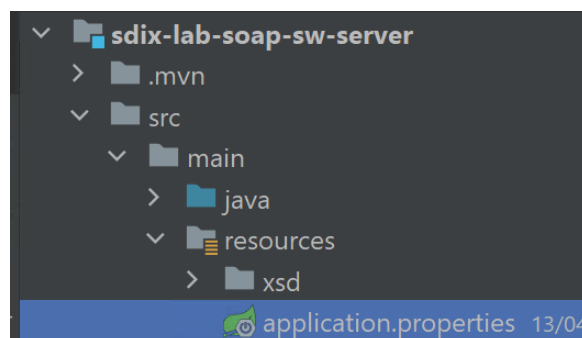
Es necesario especificar los nombres de los beans para **MessageDispatcherServlet** y **DefaultWsd11Definition**. Estos nombres son los que determinan la URL bajo la cual el servicio web y el archivo WSDL generado están disponibles. En este caso, el fichero WSDL estará disponible en

***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 que escuchará el servicio web SOAP**. Para ello, 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*** sobre la clase principal del proyecto (***SdixLabSoapSwServerApplication***), finalmente probamos en un navegador con la siguiente url y veremos que el fichero WSDL está disponible.

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

```
<?xml version='1.0' encoding='utf-8'>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:sch="http://uniovi.com/sdi/soap/ws"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://uniovi.com/sdi/soap/ws" targetNamespace="http://uniovi.com/sdi/soap/ws">
  <wsdl:types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" targetNamespace="http://uniovi.com/sdi/soap/ws">
      <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 maxOccurs="unbounded" name="mark" type="tns:mark"/>
        </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>
  </wsdl:types>
  <wsdl:message name="getMarksRequest">
    <wsdl:part element="tns:getMarksRequest" name="getMarksRequest"/>
  </wsdl:message>
  <wsdl:message name="getMarksResponse">
    <wsdl:part element="tns:getMarksResponse" name="getMarksResponse"/>
  </wsdl:message>
  <wsdl:portType name="MarksPort">
    <wsdl:operation name="getMarks">
      <wsdl:input message="tns:getMarksRequest" name="getMarksRequest"/>
      <wsdl:output message="tns:getMarksResponse" name="getMarksResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="MarksPortSoap11" type="tns:MarksPort">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getMarks">
      <soap:operation soapAction="">
        <wsdl:input name="getMarksRequest">
          <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="getMarksResponse">
          <soap:body use="literal"/>
        </wsdl:output>
      </soap:operation>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="MarksPortService">
    <wsdl:port binding="tns:MarksPortSoap11" name="MarksPortSoap11">
      <soap:address location="http://localhost:8090/webservice/marks"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

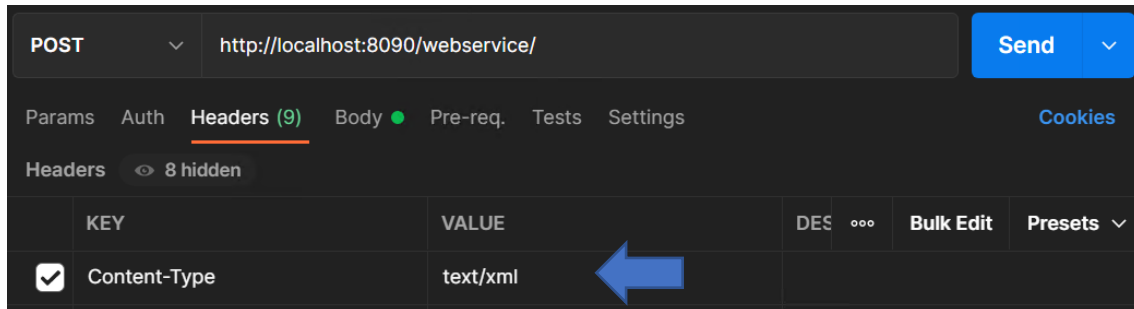
### 1.1.10 Probando el Web Service WSDL

Una vez ejecutada y desplegada la aplicación podemos probar el servicio web haciendo una petición al servicio. A continuación, **enviaremos una petición POST** para que nos devuelva las notas de un determinado alumno. Para enviar la petición, emplearemos



[Postman](#) tal y como hicimos en la práctica anterior, aunque también sería posible utilizar [SoapUI](#) u otras aplicaciones similares.

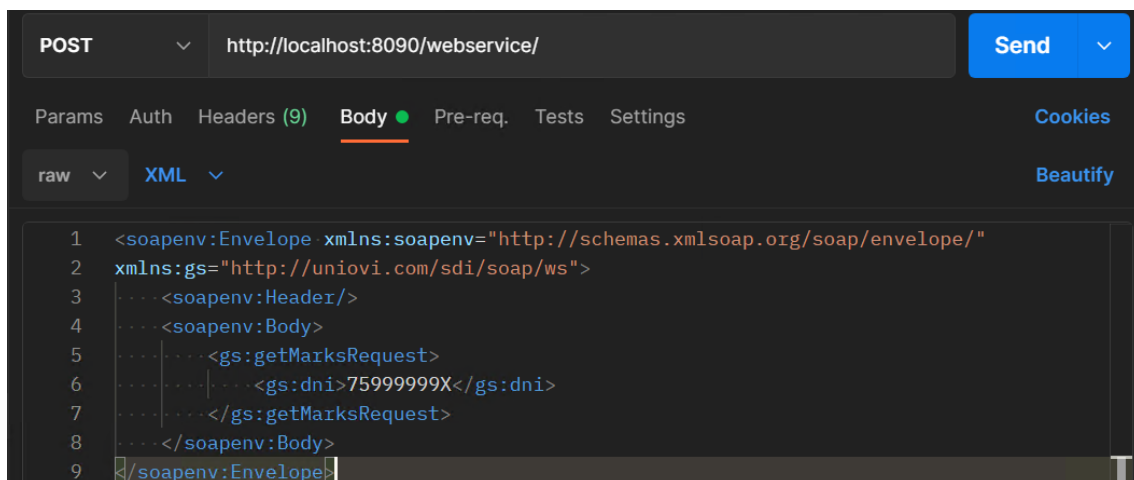
El primer paso es indicar en la **cabecera de la petición** que el **content-type** a enviar es **text/xml**.



A continuación, definimos la **petición POST** a <http://localhost:8090/webservice/> con un **sobre (soapenv:Envelope)**, en el cuerpo del mensaje, que solicita las notas del alumno con **DNI 75999999X**

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

Esta sería la petición desde Postman:





Respuesta recibida:

```
Body 200 OK 11 ms 808 B Save Response
Pretty Raw Preview Visualize XML
1 <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
2   <SOAP-ENV:Header/>
3   <SOAP-ENV:Body>
4     <ns2:getMarksResponse xmlns:ns2="http://uniovi.com/sdi/soap/ws">
5       <ns2:user>
6         <ns2:dni>75999999X</ns2:dni>
7         <ns2:name>Jose</ns2:name>
8         <ns2:mark>
9           <ns2:description>SDI</ns2:description>
10          <ns2:score>10</ns2:score>
11        </ns2:mark>
12        <ns2:mark>
13          <ns2:description>DLP</ns2:description>
14          <ns2:score>8</ns2:score>
15        </ns2:mark>
16        <ns2:mark>
17          <ns2:description>IP</ns2:description>
18          <ns2:score>8</ns2:score>
19        </ns2:mark>
20      </ns2:user>
21    </ns2:getMarksResponse>
22  </SOAP-ENV:Body>
23 </SOAP-ENV:Envelope>
```

**Nota:** Si al realizar la petición al servicio web nos devuelve el siguiente error entonces debemos incluir en el POM las dependencias que se indican a continuación:

Posible error:

```
Pretty Raw Preview Visualize XML
1 <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
2   <SOAP-ENV:Header/>
3   <SOAP-ENV:Body>
4     <SOAP-ENV:Fault>
5       <faultcode>SOAP-ENV:Server</faultcode>
6       <faultstring xml:lang="en">Implementation of JAXB-API has not been found on
7         module path or classpath.</faultstring>
8     </SOAP-ENV:Fault>
9   </SOAP-ENV:Body>
10 </SOAP-ENV:Envelope>
```

Dependencias:



```
<dependency>  
  <groupId>javax.xml.bind</groupId>  
  <artifactId>jaxb-api</artifactId>  
</dependency>  
<dependency>  
  <groupId>javax.activation</groupId>  
  <artifactId>activation</artifactId>  
  <version>1.1</version>  
</dependency>  
<dependency>  
  <groupId>org.glassfish.jaxb</groupId>  
  <artifactId>jaxb-runtime</artifactId>  
</dependency>
```

Para finalizar este apartado, subimos el proyecto a GitHub y hacemos el siguiente commit:

**Nota: Incluir el siguiente Commit Message ->**

**“SDI-IDGIT-SW-11.1-Servicio-SOAP-Server”**

**OJO: sustituir IDGIT por tu número asignado (p.e. 2324-101):**

**“SDI-2324-101-SW-11.1-Servicio-SOAP-Server.”**

**(No olvides incluir los guiones y NO incluyas BLANCOS al principio o al final,  
ni las comillas)**

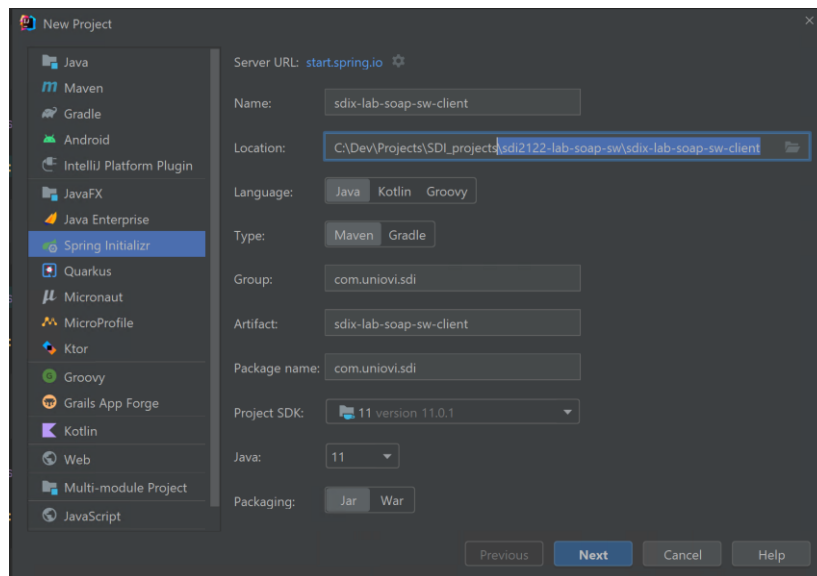


## 1.2 Consumiendo un servicio web

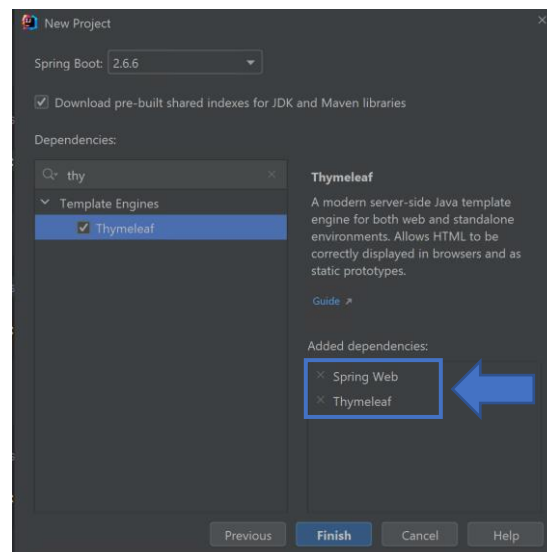
En esta segunda parte, construiremos una aplicación web cliente que consumirá el servicio web SOAP creado en el paso anterior.

Este proyecto lo guardaremos en el mismo **repositorio en GitHub que creamos en la primera parte, es decir, en el repositorio con el nombre sdix-lab-soap-ws** (sustituyendo la x por el IDGIT correspondiente).

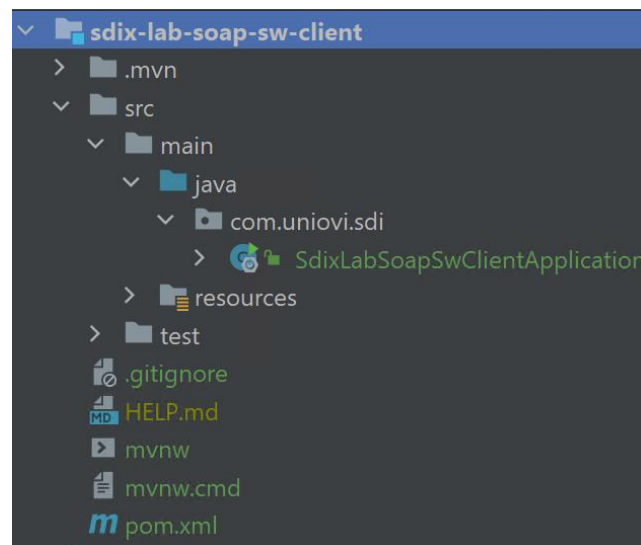
En el IntelliJ IDEA creamos un nuevo proyecto Spring Boot que llamaremos **sdix-lab-soap-sw-client** (sustituyendo la x por el IDGIT correspondiente, desde el **menú *File /New /Project / Spring Initializr*** y completamos los datos similares a los que se muestran en la siguiente imagen:



A continuación, en la siguiente pantalla **indicamos que utilizaremos las dependencias Spring Web y Thymeleaf** y finalizamos el proceso:



A continuación, abrimos el proyecto en una nueva instancia de IntelliJ IDEA y con esto ya tendremos el proyecto creado como se muestra la siguiente imagen:



### 1.2.1 Añadir la dependencia Spring-WS

Para poder trabajar con servicios web SOAP es necesario añadir la dependencia 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 a partir de un WSDL** que, en nuestro caso, lo obtenemos de un servicio web SOAP. Similar al caso anterior, para generar las clases JAVA en base a un WSDL, utilizaremos el plugin **maven-jaxb2-plugin**.

Java Architecture for XML Binding (JAXB) proporciona la capacidad de serializar y deserializar las referencias de objetos Java a XML y viceversa. Para lograr estos objetivos es necesario incluir las siguientes dependencias en el fichero **POM.XML** del proyecto:

```
<dependency>
  <groupId>com.sun.xml.bind</groupId>
  <artifactId>jaxb-core</artifactId>
  <version>2.3.0.1</version>
</dependency>
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
</dependency>
<dependency>
  <groupId>com.sun.xml.bind</groupId>
  <artifactId>jaxb-impl</artifactId>
  <version>2.3.1</version>
</dependency>
```

A continuación, incluimos el plugin en nuestro proyecto añadiendo las siguientes referencias en el POM:

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



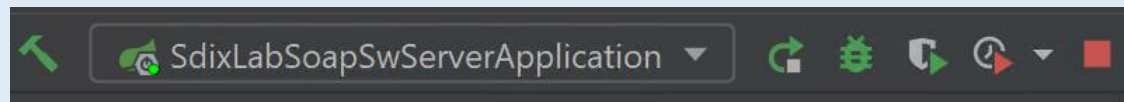


```
</schema>  
</schemas>  
</configuration>  
</plugin>
```

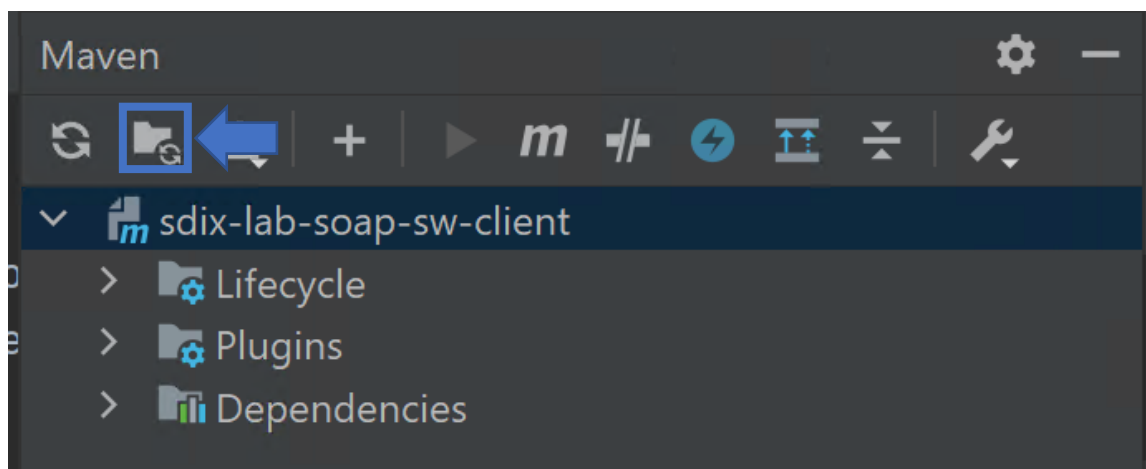
Sobre el código que acabamos de añadir, destacamos los siguientes elementos:

- **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 especifican las diferentes localizaciones (URL) de los ficheros WSDL expuestos mediante servicios web SOAP.

**Nota:** Al incluir el plugin, no se generarán las clases automáticamente, debido a que el plugin está intentando acceder al fichero WSDL localizado en la URL <http://localhost:8090/webService/marks.wsdl>. Esto se debe a que el servicio web SOAP no está disponible. Para solucionarlo solo tenemos que desplegar la aplicación del servicio web SOAP definido en el proyecto anterior (**sdix-lab-soap-sw-server**). Solo tenemos que ejecutar la aplicación *desde otra instancia de IntelliJ IDEA*.



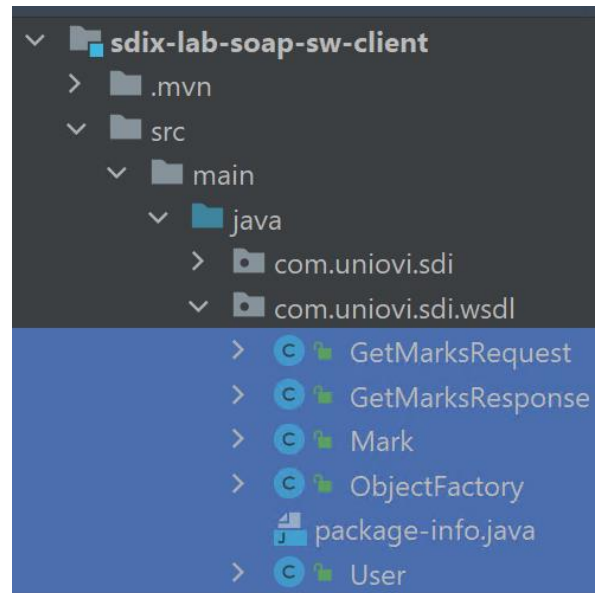
A continuación, volvemos al proyecto cliente y actualizamos todos los recursos de proyecto. Vamos panel de **Maven** y pulsamos el botón **Actualizar all projects**:







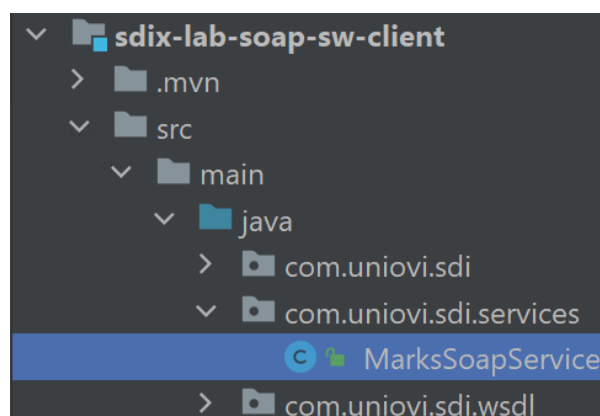
Finalmente, hacemos click derecho sobre el nombre del proyecto y pulsamos la opción **Reload From Disk Project**. Esto mostrará en el proyecto todas las clases **JAVA** generadas apartir de la llamada al **WSDL**, como se muestra en la siguiente imagen:



### 1.2.3 Crear un servicio web cliente

En este apartado, crearemos un servicio web cliente para obtener las notas de los alumnos consumiendo el servicio web SOAP publicado anteriormente.

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.uniovi.sdi.services** y dentro la clase **MarksSoapService**.



El contenido de la clase MarksSoapService será el siguiente:

```
package com.uniovi.sdi.services;
```



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

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);
        return (GetMarksResponse)
getWebServiceTemplate().marshalSendAndReceive(serviceEndpoint,
        request, new SoapActionCallback(serviceSoapAction));
    }
}
```

Este servicio 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** generadas a partir del WSDL mediante **JAXB**.

#### 1.2.4 Modificar el fichero application.properties

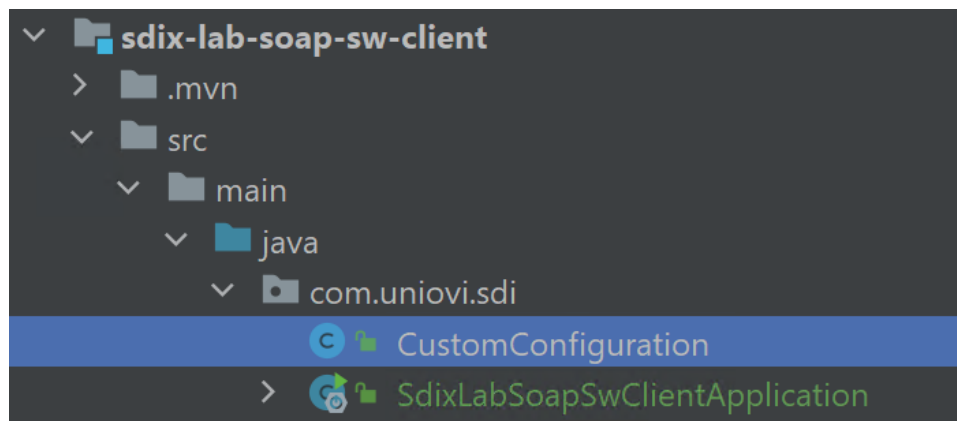
En el fichero de configuración **application.properties** definimos el **puerto** por el que escuchará la aplicación web cliente, así como las variables del **endpoint** y **el método** que se inyectará en diferentes puntos de la aplicación.

```
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 una petición a un servicio web SOAP, la respuesta es devuelta en formato XML. Por tanto, necesitamos serializar y deserializar las peticiones que se realizan. Spring WS utiliza el módulo **OXM** de Spring Framework que tiene el **Jaxb2Marshaller** para serializar y deserializar solicitudes XML.

Para llevar a cabo este proceso, creamos una nueva clase **CustomConfiguration** que incluya los beans necesarios para serializar las peticiones.



```
package com.uniovi.sdi;

import com.uniovi.sdi.services.MarksSoapService;
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;

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

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

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



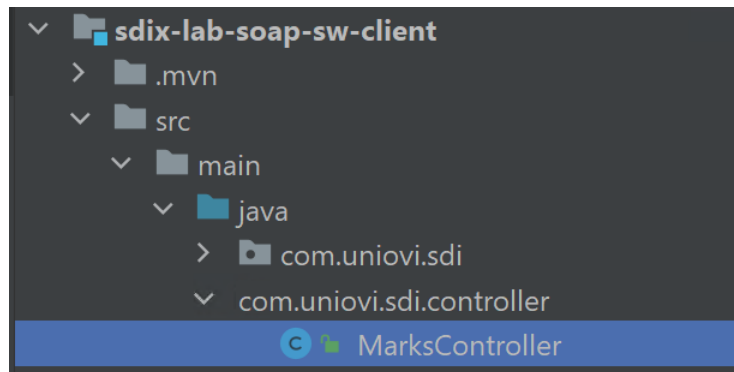
El **marshaller** apunta a la colección de objetos de dominio generados y los utilizará para serializar y deserializar entre **XML** y **POJOs**.

El **marksService** se crea y configura con la URI del servicio de notas definido 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.

**Creemos el paquete *com.uniovi.sdi.controllers* y dentro de este la clase *MarksController***. Este controlador contiene el método ***getMarks()*** que recibirá, a través de una petición GET, el **DNI** de un usuario y devolverá la lista de notas del usuario.



El contenido de la clase MarksController será el siguiente:

```
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import java.util.ArrayList;
import java.util.List;

@Controller
public class MarksController {

    private final MarksSoapService marksSoapService;

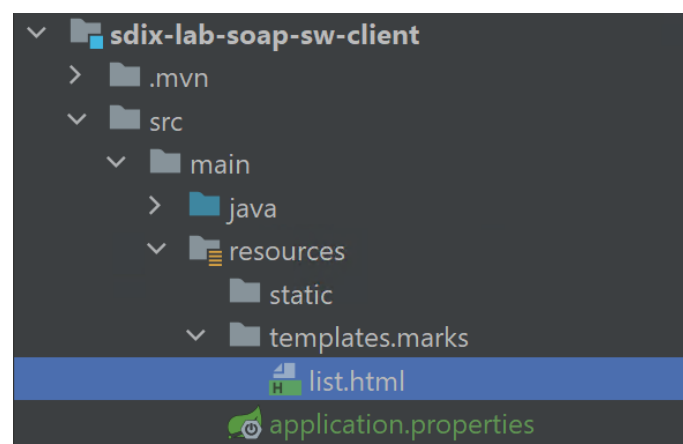
    public MarksController(MarksSoapService marksSoapService) {
        this.marksSoapService = marksSoapService;
    }
}
```



```
@RequestMapping("/marks/list")
public String getMarks(Model model, @RequestParam String dni) {
    List<Mark> marks = new ArrayList<>();
    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 devuelve el controlador, en formato HTML. En este caso se mostrará la lista de notas de un usuario específico. En la carpeta **resources/templates** creamos una carpeta **marks** y dentro un fichero **list.html**.



El contenido del fichero HTML será el siguiente:

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org">
<head>
    <title>SDILab11 - cliente SOAP</title>
    <meta charset="utf-8"/>
```

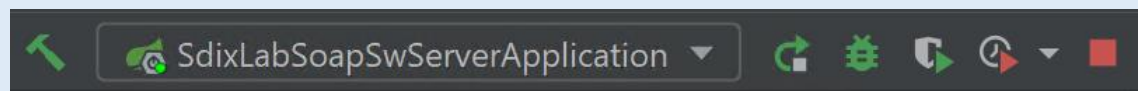


```
</head>
<body>
<div class="container">
  <h2>
    Listado de notas de: <span th:text="{dni}">99999999K</span>
  </h2>
  <table>
    <thead>
      <tr>
        <th>Descripción</th>
        <th>Puntuación</th>
      </tr>
    </thead>
    <tbody>
      <tr th:each="mark : {markList}">
        <td th:text="{mark.description}">asignatura</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 (***sdix-lab-soap-ws-client***) e introducimos en el navegador la siguiente URL: ***http://localhost:8091/marks/list?dni=75999999X***

**Nota 1:** Es necesario que el servicio web servidor esté disponible, por lo que debemos asegurarnos de que la aplicación ***sdix-lab-soap-sw-server*** esté en ejecución.



**Nota 2:** Si estamos usando una versión igual o superior a java 11, posiblemente el proyecto dará el siguiente error, ya que las librerías SOAP de Java EE se eliminaron en Java 11.



```
Sync × Build Output ×
! sdix-lab-soap-sw-client: build failed At 18/04/2022 9:55 with 42 errors 7 sec, 610 ms
GetMarksResponse.java src\main\java\com\uniovi\sdi\wsdl 10 errors
! package javax.xml.bind.annotation does not exist :11
! package javax.xml.bind.annotation does not exist :12
! package javax.xml.bind.annotation does not exist :13
! package javax.xml.bind.annotation does not exist :14
! package javax.xml.bind.annotation does not exist :15
! cannot find symbol class XmlAccessorType :37
```

Para corregir estos errores añadimos las siguientes dependencias al POM y luego las recargamos desde el panel de Maven:

```
<dependency>
  <groupId>jakarta.xml.ws</groupId>
  <artifactId>jakarta.xml.ws-api</artifactId>
</dependency>
<dependency>
  <groupId>jakarta.xml.bind</groupId>
  <artifactId>jakarta.xml.bind-api</artifactId>
</dependency>
<dependency>
  <groupId>jakarta.xml.soap</groupId>
  <artifactId>jakarta.xml.soap-api</artifactId>
</dependency>
<dependency>
  <groupId>com.sun.xml.messaging.saaj</groupId>
  <artifactId>saaj-impl</artifactId>
</dependency>
```

Si ahora hacemos una petición con un dni válido desde un navegador web, veremos que el servicio nos devuelve la lista de notas para ese usuario en concreto:

SDILab11 - cliente SOAP

localhost:8091/marks/list/?dni=75999999X

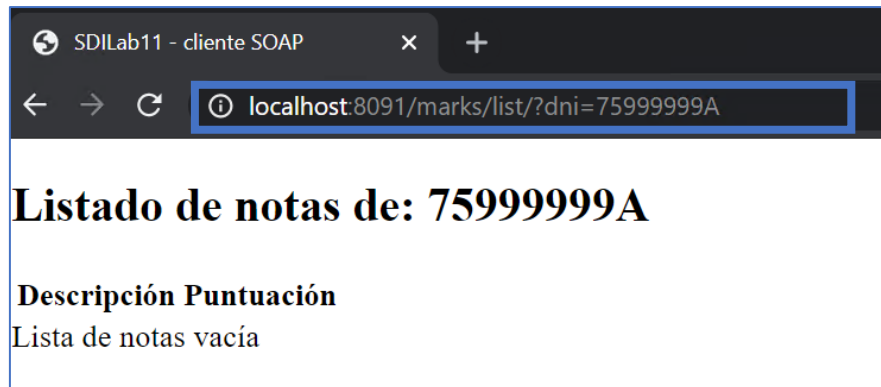
### Listado de notas de: 75999999X

Descripción	Puntuación
SDI	10
DLP	8
IP	8





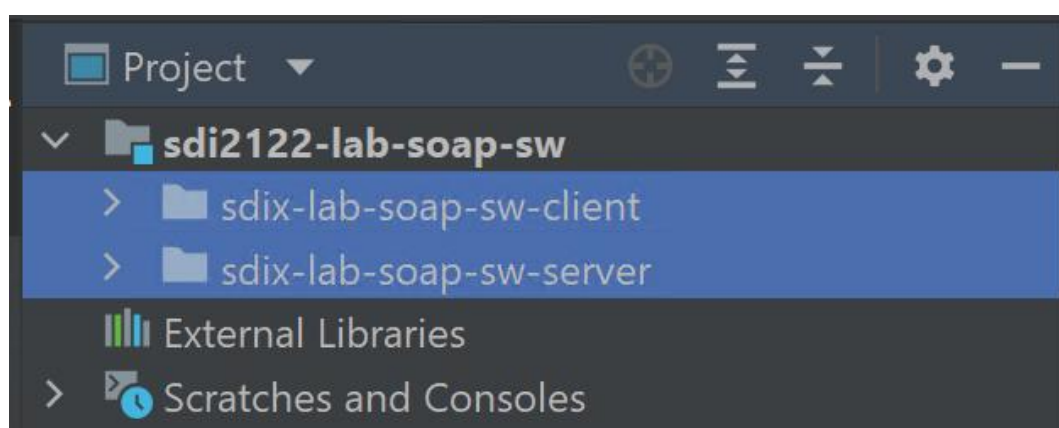
Sin embargo, si usamos un dni inexistente, el servicio nos devolverá una lista vacía.



Para finalizar este apartado, subimos el proyecto a GitHub y hacemos el siguiente commit:

**Nota: Incluir el siguiente Commit Message ->**  
**“SDI-IDGIT-SW-11.1- Servicio SOAP Cliente”**  
**OJO: sustituir IDGIT por tu número asignado (p.e. 2324-101):**  
**“SDI-2324-101-SW-11.1- Servicio SOAP Cliente.”**  
(No olvides incluir los guiones y NO incluyas BLANCOS al principio o al final,  
ni las comillas)

Si abrimos el proyecto raíz en IntelliJ IDEA deberíamos tener dos proyectos como se muestran a continuación:










## 1.3 Resultado esperado en los repositorios de GitHub

### Proyectos:

	edwardnunez SDI-2122-101-SW-11.1-Servicio SOAP Cliente.	21865b0 4 minutes ago
	sdix-lab-soap-sw-client	SDI-2122-101-SW-11.1-Servicio SOAP Cliente.
	sdix-lab-soap-sw-server	SDI-2122-101-SW-11.1-Servicio SOAP-Server.

### Commits:

- ⇒ SDI-2324-101-SW-11.1-Servicio SOAP-Server.
- ⇒ SDI-2324-101-SW-11.1-Servicio SOAP Cliente