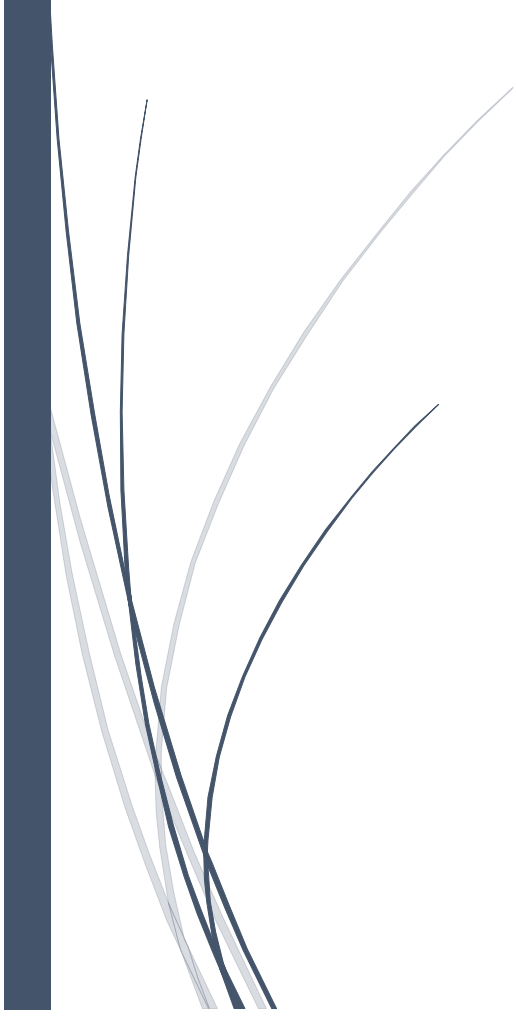
TP : Développer un service web SOAP avec JAX WS



Architecture des composants d’entreprise

Table des matières

I. Objectif du TP

- Comprendre l'architecture du standard ***SOAP***.
- Comprendre le fichier ***WSDL***.
- Comprendre le style ***Document*** des messages SOAP.
- Développer le WS avec l'api ***JAX-WS***.
- Générer le Stub pour consommer le WS en utilisant le plugin ***« Jakarta EE : Web Services (JAX-WS) »*** d'Intellig version Ultimate.
- Générer le Stub pour consommer le WS en utilisant le plugin ***jaxws-maven-plugin*** de Maven.

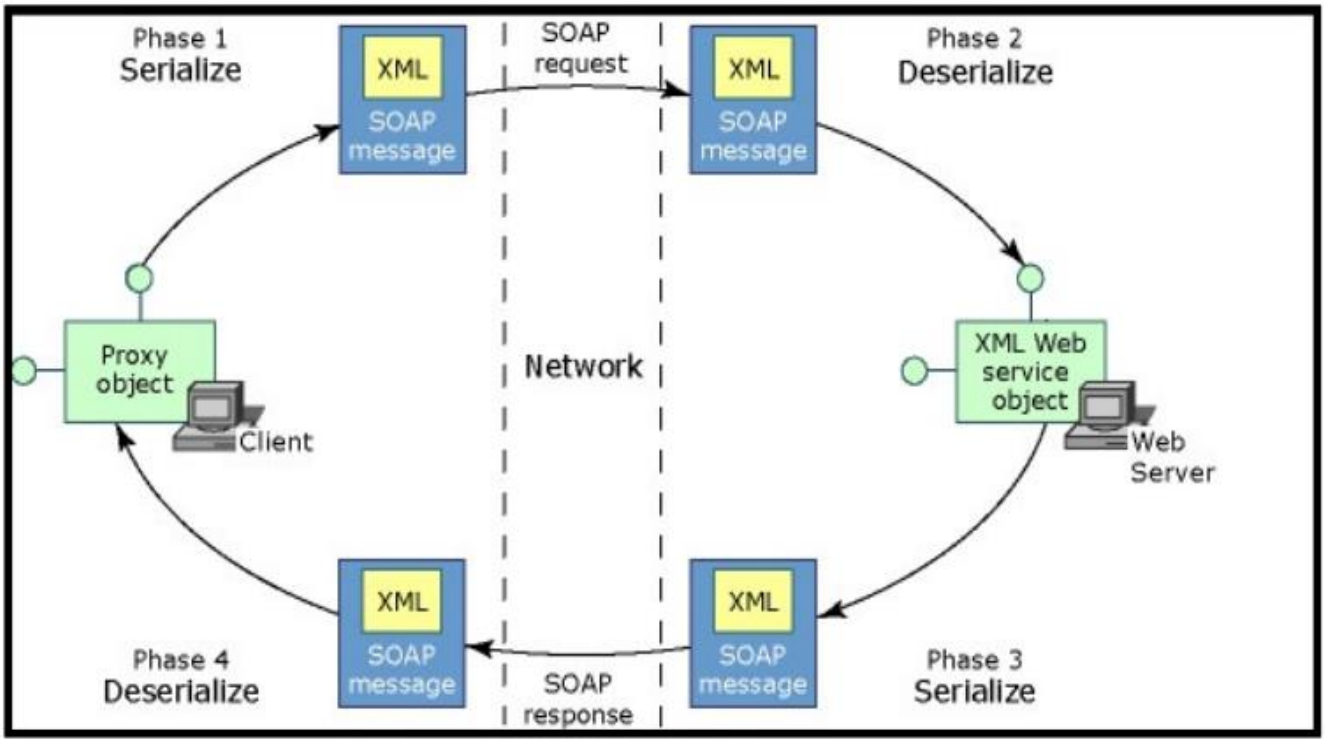
II. Prérequis

- IntelliJ IDEA ;
- JDK version 17 ;
- Une connexion Internet pour permettre à Maven de télécharger les librairies.
- L'outil SoapUI pour tester le WS (https://www.soapui.org/downloads/soapui/).

NB : *Ce TP a été réalisé avec IntelliJ IDEA 2023.2.3 (Ultimate Edition).*

III. L'architecture SOAP

a. Présentation des SW avec SOAP



L'appel à un service web de type SOAP suit plusieurs étapes :

1. Le client instancie une classe de type proxy encapsulant le service Web XML.
2. Le client invoque une méthode du proxy.

3. Le moteur SOAP sur le client crée le message à partir des paramètres utilisés pour invoquer la méthode

4. Le moteur SOAP envoie le message SOAP au serveur généralement en utilisant le protocole HTTP

5. Le moteur SOAP du serveur réceptionne et analyse le message SOAP

6. Le moteur fait appel à la méthode de l'objet correspondant à la requête SOAP

7. Le moteur SOAP sur le serveur crée le message réponse à partir de la valeur de retour

8. Le moteur SOAP envoie le message SOAP contenant la réponse au client généralement en utilisant le protocole http

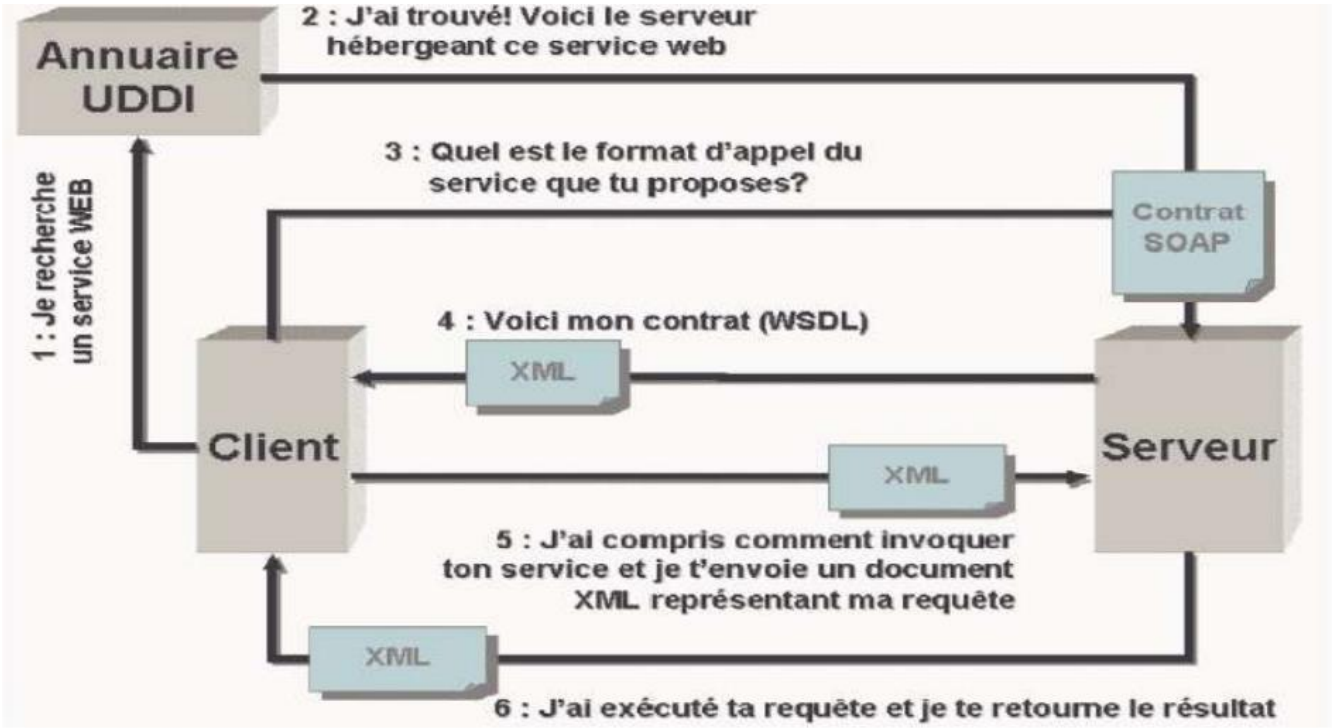
9. Le moteur SOAP du client réceptionne et analyse le message SOAP

10. Le moteur SOAP du client instancie un objet à partir du message SOAP

b. L'architecture SOAP

L'architecture des services web est composée de quatre grandes couches utilisant plusieurs technologies :

- ✓ ***Découverte*** : cette couche représente un annuaire dans lequel il est possible de publier des services et de les rechercher : ***UDDI*** (Universal Description, Discovery, and Integration) est le standard.
- ✓ ***Description*** : cette couche normalise la description de l'interface publique d'un service web en utilisant ***WSDL*** (Web Service Description Language).
- ✓ ***Communication*** : cette couche permet d'encoder les messages échangés (***SOAP*** est le standard).
- ✓ ***Transport*** : cette couche assure le transport des messages : généralement HTTP est mis en œuvre mais d'autres protocoles peuvent être utilisés (SMTP, FTP, ...).



c. SOAP

- SOAP (Simple Object Access Protocol) est un standard du W3C qui permet l'échange formaté d'informations entre un client et un serveur. SOAP peut être utilisé pour la requête et la réponse de cet échange.
- SOAP assure la partie messaging dans l'architecture des services web : il est utilisé pour normaliser le format des messages échangés entre le consommateur et le fournisseur de services web.
- SOAP est un protocole qui est principalement utilisé pour dialoguer avec des objets distribués.
- Son grand intérêt est d'utiliser XML ce qui le rend ouvert contrairement aux autres protocoles qui sont propriétaires : cela permet la communication entre un client et un serveur utilisant des technologies différentes. SOAP fait un usage intensif des espaces de nommages (namespaces).
- SOAP est défini pour être indépendant du protocole de transport utilisé pour véhiculer le message. Cependant, le protocole le plus utilisé avec SOAP est HTTP. Son utilisation avec SOAP permet de rendre les services web plus interopérables.
- D'autres protocoles peuvent être utilisés (par exemple SMTP ou FTP).
- SOAP est aussi indépendant de tout système d'exploitation et de tout langage de programmation car il utilise XML. Ceci permet une exposition et une consommation de services web avec des outils et des OS différents.
- SOAP peut être utilisé pour :
    - Appeler une méthode d'un service (SOAP RPC)
    - Echanger un message avec un service (SOAP Messaging)¬
    - Recevoir un message d'un service

d. Les formats du message SOAP

Deux formats de messages SOAP sont définis (Ce qu'on appelle le Style):

- **Remote Procedure Call (RPC) :** permet l'invocation d'opérations qui peuvent retourner un résultat. Les messages contiennent le nom de l'opération, les paramètres en entrée multiples et valeur de retour.
- **Message Oriented (Document)** : données au format XML définies dans un schéma XML. Les messages *ne contiennent pas le nom de l'opération*, un document XML en entrée et en retour

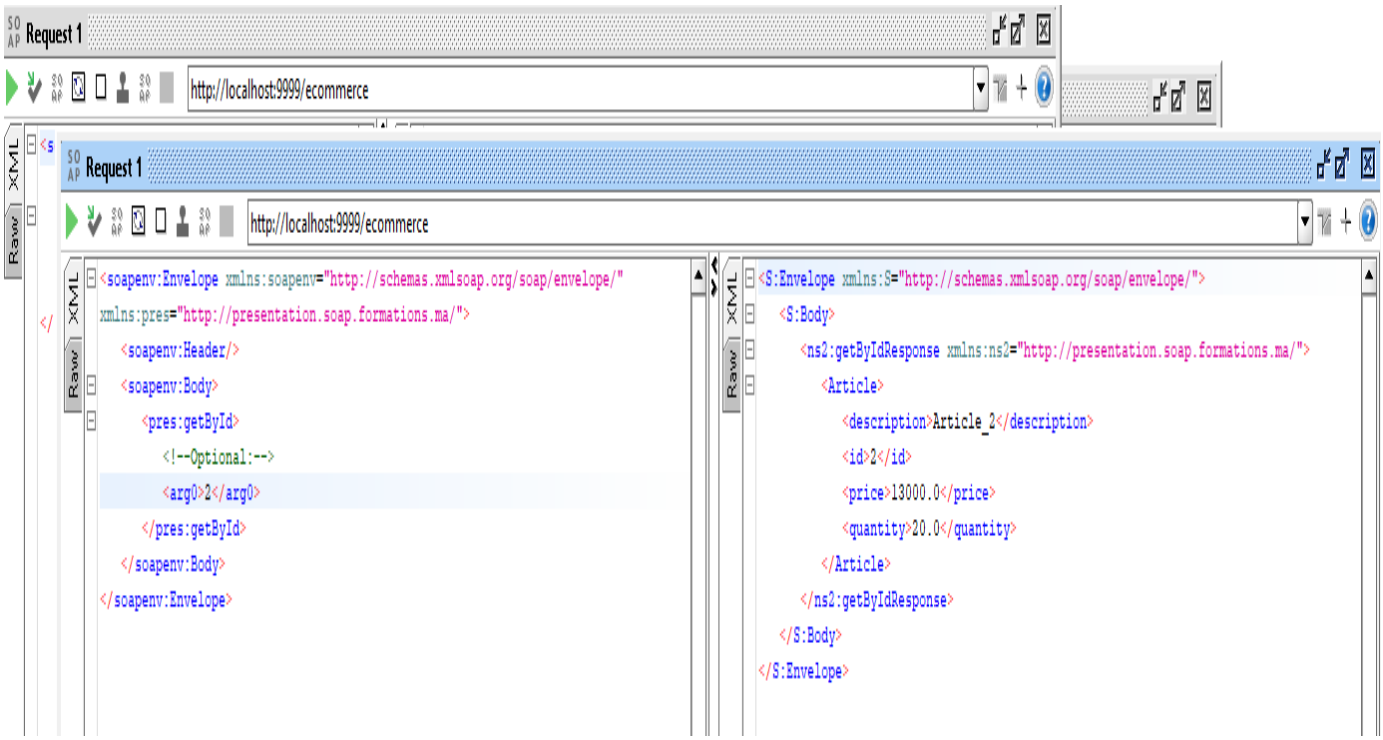
Les règles d'encodage (***Encoding rules***) précisent les mécanismes de sérialisation des données dans un message. Il existe deux types :

- **Encoded** : les paramètres d'entrée de la requête et les données de la réponse sont encodées en XML dans le corps du message selon un format particulier à SOAP.

- **Literal** : les données n'ont pas besoin d'être encodées de façon particulière : elles sont directement encodées en XML selon un schéma défini dans le WSDL.

- Le style (RPC ou Document) et le type d'encodage (Encoded ou Literal) permettent de définir comment les données seront sérialisées et désérialisées dans les requêtes et les réponses.
- La combinaison du style et du type d'encodage peut prendre plusieurs valeurs :
  - **RPC/Encoded**
  - **RPC/Literal**
  - **Document /Encoded**
  - **Document/Literal**
  - **Wrapped Document/Literal** : extension du Document/Literal proposé par Microsoft.
- Le style **RPC/Encoded** a largement été utilisé au début des services web : actuellement ce style est en cours d'abandon par l'industrie au profit du style **Document/Literal**.
- Le style et le type d'encodage sont précisés dans le WSDL. L'appel du service web doit obligatoirement se faire dans le style précisé dans le WSDL puisque celui-ci détermine le format des messages échangés.

- Exemple de requête/réponse SOAP avec le style Document :



e. WSDL

- ***WSDL (Web Service Description Language)*** est utilisé pour fournir une description d'un service web afin de permettre son utilisation. C'est une recommandation du W3C.
- Pour permettre à un client de consommer un service web, ce dernier a besoin d'une description détaillée du service avant de pouvoir interagir avec lui. Un WSDL fournit cette description dans un document XML. WSDL joue un rôle important dans l'architecture des Services Web en assurant la partie description : il contient toutes les informations nécessaires à l'invocation du service qu'il décrit.
- La description WSDL d'un service web comprend une définition du service, les types de données utilisés notamment dans le cas de types complexes, les opérations utilisables, le protocole utilisé pour le transport et l'adresse d'appel.
- C'est un document XML qui décrit un service web de manière indépendante de tout langage. Il permet l'appel de ses opérations et l'exploitation des réponses (les paramètres, le format des messages, le protocole utilisé, ...)
- WSDL est conçu pour être indépendant de tout protocole. Comme SOAP et HTTP sont les deux protocoles les plus couramment utilisés pour implémenter les services web, le standard WSDL intègre un support de ces deux protocoles.
- L'utilisation de XML permet à des outils de différents systèmes, plateformes et langages d'utiliser le contenu d'un WSDL pour générer du code permettant de consommer un service web.
- Les moteurs SOAP proposent en général un outil qui va lire le WSDL et générer les classes requises pour utiliser un service web avec la technologie du moteur SOAP. Le code généré utilise un moteur SOAP qui masque toute la complexité du protocole utilisé et des messages échangés lors de la consommation de services web en agissant comme un proxy. Par exemple, Axis propose l'outil WSDL2Java pour la génération de ces classes à partir du WSDL.
- Les spécifications de WSDL sont consultables à l'url http://www.w3.org/standards/techs/wsdl#w3c_all.
- Un document WSDL définit plusieurs éléments :

Type	La définition des types de données utilisés.
Message	La définition de la structure d'un message en lui attribuant un nom et en décrivant les éléments qui le composent avec un nom et un type.
PortType	La description de toutes les opérations proposées par le service web (interface du service) et identification de cet ensemble avec un nom.
Operation	La description d'une action proposée par le service web notamment en précisant les messages en entrée (input) et en sortie (output).
Binding	La description du protocole de transport et d'encodage utilisé par un PortType afin de

	pouvoir invoquer un service web.
Port	Référence un Binding (généralement cela correspond à l'url d'invocation du service web)
Service	C'est un ensemble de ports.

f. Les conseils pour la mise en œuvre

- Avant de développer des services web, il faut valider la solution choisie avec un POC (Proof Of Concept) ou un prototype. Lors de ces tests, il est important de vérifier l'interopérabilité notamment si les services web sont consommés par différentes technologies.
- Le choix du moteur SOAP est aussi très important notamment vis-à-vis du support des spécifications, des performances, de la documentation, …

g. Les étapes de mise en œuvre

Etape 1 : Définition des contrats des services métiers

Cette étape est une phase d'analyse qui va définir les fonctionnalités proposées par chaque service pour répondre aux besoins.

Etape 2 : Identification des services web

- Cette étape doit permettre de définir les contrats techniques des services web à partir des services métiers définis dans l'étape précédente.
- Un service métier peut être composé d'un ou plusieurs services web.
- La réalisation de cette étape doit tenir compte de plusieurs contraintes :
  - ✓ Penser forte granularité / faible couplage
  - ✓ Tenir compte de contraintes techniques
  - ✓ Préférer les services web indépendants du contexte client.
- L'invocation d'un service est coûteuse notamment à cause du mapping objet/xml et xml/objet réalisé à chaque appel. Cette règle est vraie pour toutes les invocations de fonctionnalités distantes mais encore plus avec les services web. Il est donc préférable de limiter les invocations de méthodes d'un service web en proposant des fonctionnalités à forte granularité.

  Par exemple, il est préférable de définir une opération qui permet d'obtenir les données d'une entité plutôt que de proposer autant d'opérations que l'entité possède de champs. Ceci permet de réduire le nombre d'invocations du service web et réduit le couplage entre la partie front-end et back-end.
- La définition des services web doit tenir compte de contraintes techniques liées aux performances ou à la consommation de ressources. Par exemple, si le temps de traitement d'un service web est long, il

faudra prévoir son invocation de façon asynchrone ou si les données retournées sont des binaires de tailles importantes, il faudra envisager d'utiliser le mécanisme de pièces jointes (attachment).

- Il est préférable de définir des services web qui soient Stateless (ne reposant pas par exemple sur une utilisation de la session http). Ceci permet de déployer les services web dans un cluster où la réplication de session sera inutile.

Etape 3 : écriture des services web

- Cette étape est celle du codage proprement dit des services web.
- Deux approches sont possibles :
  - ✓ écriture du WSDL en premier (***contract first***) : des outils du moteur SOAP sont utilisés pour gérer le code des services web à partir du WSDL. Face à la complexité de la rédaction du WSDL, cette approche n'est pas toujours privilégiée.
  - ✓ écriture de la classe et génération du WSDL (***code first***) : chaque service est implémenté sous la forme d'une ou plusieurs classes et c'est le moteur SOAP utilisé qui va générer le WSDL correspondant en se basant sur la description de la classe et des métadonnées.

Etape 4 : déploiement et tests

- Les services web doivent être packagés et déployés généralement dans un serveur d'applications ou un conteneur web.
- Pour tester les services web, il est possible d'utiliser des outils fournis par l'IDE ou d'utiliser des outils tiers comme ***SoapUI*** qui propose de très nombreuses fonctionnalités pour les tests des services web allant de la simple invocation à l'invocation de scénarios complexes et de tests de charges.

Etape 5 : consommation des services web par les applications clientes

- Il faut mettre en œuvre les outils du moteur SOAP utilisé par l'application cliente pour générer les classes nécessaires à l'invocation des services web et utiliser ces classes dans l'application. C'est généralement le moment de faire quelques adaptations pour permettre une bonne communication entre le client et le serveur.

h. Quelques recommandations

- Il ne faut pas se lier à un langage de programmation :
  - ✓ N'utiliser que des types communs : int, float, String, Date, ...
  - ✓ Ne pas utiliser de types spécifiques : Object, DataSet, ...
  - ✓ Eviter la composition d'objets.

- ✓ Utiliser un tableau ou des collections typées avec un generic plutôt qu'une collection non typée.

- Il faut éviter la surcharge des méthodes.
- Il faut éviter de transformer une classe en service web (notamment en utilisant des annotations) : il est recommandé de définir une interface qui va établir le contrat entre le service et son implémentation. Cette pratique venant de la POO doit aussi s'appliquer pour les services web.

i. Les implémentations de SOAP

Il existe de nombreuses implémentations possibles de moteurs SOAP permettant la mise en œuvre de services web avec Java, notamment plusieurs solutions open source :

- ✓ Intégrées à la plate-forme Java EE 5.0 et Java SE 6.0. Il est à préciser qu'à partir de JDK 9, toutes les spécifications de la communauté Jakarta EE (javax.*), notamment JAX WS, ont été supprimées de la JDK.
- ✓ JWSDP de Sun
- ✓ Axis et Axis 2 (Apache eXtensible Interaction System) du projet Apache
- ✓ XFire
- ✓ CXF du projet Apache
- ✓ JBoss WS
- ✓ Metro du projet GlassFish

IV. Partie 1 : Développement du WS (le serveur)

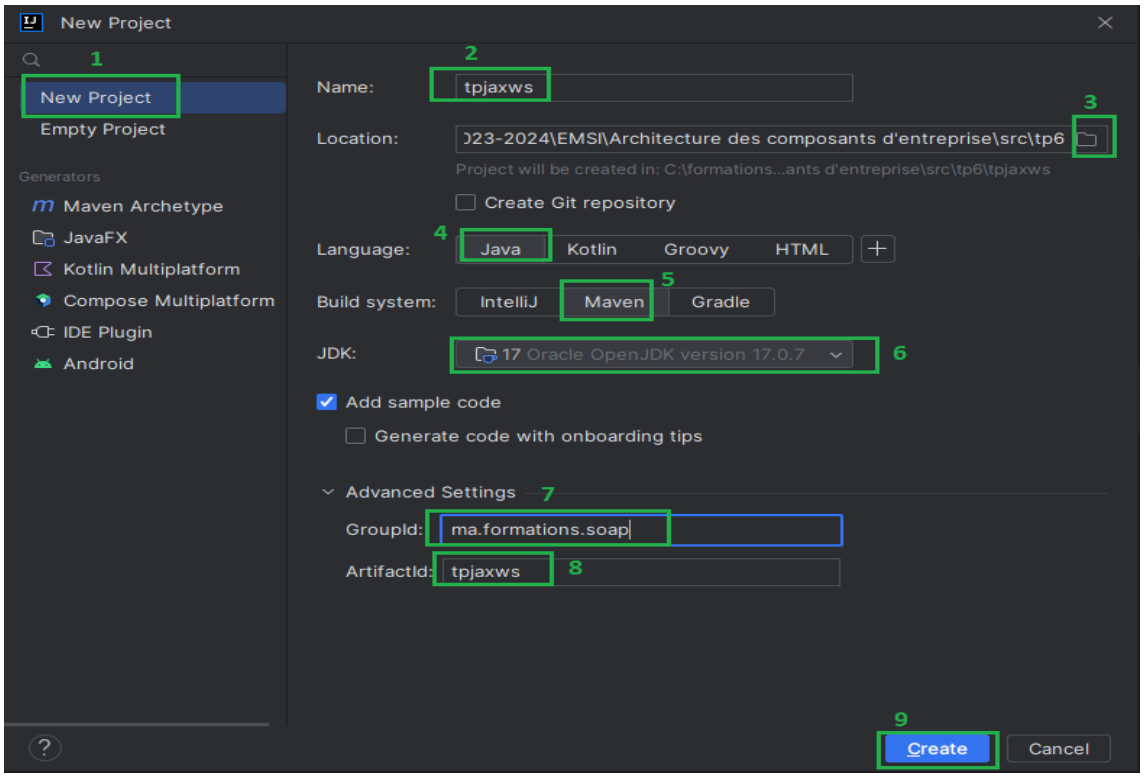
a. Objectif

Le service web que nous allons développer offre les opérations suivantes :

- Récupérer la liste des articles.
- Récupérer un article par son ID.
- Sauvegarder un article (ajout ou modification).
- Supprimer un article par son ID.

b. Création du projet Maven

- Créer un nouveau projet Maven comme expliqué ci-après :



1. Cliquer sur *New Project*.
2. Entrer le nom de votre projet (par exemple : tpjaxws)
3. Préciser le chemin de votre projet (par exemple : c:\workspace)
4. Cliquer sur *Java*.
5. Cliquer sur *Maven*.
6. Préciser JDK 17.
7. Entrer le *GroupId* (par exemple : ma.formations.soap).
8. Entrer *l'ArtifactId* (par exemple : tpjaxws).
9. Cliquer sur le bouton *Create*.

c. Le fichier pom.xml

- Ajouter les dépendances suivantes au fichier pom.xml :

```
<dependency>
  <groupId>com.sun.xml.ws</groupId>
  <artifactId>jaxws-ri</artifactId>
  <version>4.0.2</version>
  <type>pom</type>
</dependency>

<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.30</version>
```

```
    <scope>provided</scope>
</dependency>
```

Le contenu global de votre fichier pom.xml est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>ma.formations.soap</groupId>
    <artifactId>tpjaxws</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.sun.xml.ws</groupId>
            <artifactId>jaxws-ri</artifactId>
            <version>4.0.2</version>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.30</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

d. Le modèle

- Créer le package *ma.formations.soap.service.model* et en suite créer la classe **Article** suivante :

```
package ma.formations.soap.service.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
```

```
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Article {
   private Long id;
   private String description;
   private Double price;
   private Double quantity;
}
```

e. La couche service

- Dans le package *ma.formations.soap.service* créer l'interface **IService** et la classe **ServiceImpl** suivantes :

L'interface IService :

```
package ma.formations.soap.service;

import ma.formations.soap.service.model.Article;
import java.util.List;

public interface IService {
   List<Article> getAll();
   Article save(Article article);
   void deleteById(Long id);
   Article getById(Long id);
}
```

La classe ServiceImpl :

```
package ma.formations.soap.service;

import ma.formations.soap.service.model.Article;

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

public class ServiceImpl implements IService {
   private static final List<Article> repository = new ArrayList<>();

   static {
```

```
repository.add(Article.builder().id(1l).description("Article_1").price(12000.0).quantity(10.0).build());
repository.add(Article.builder().id(2l).description("Article_2").price(13000.0).quantity(20.0).build());
repository.add(Article.builder().id(3l).description("Article_3").price(13000.0).quantity(30.0).build());
repository.add(Article.builder().id(4l).description("Article_4").price(14000.0).quantity(40.0).build());
repository.add(Article.builder().id(5l).description("Article_5").price(15000.0).quantity(50.0).build());
    }

    @Override
    public List<Article> getAll() {
        return repository;
    }

    @Override
    public Article save(Article article) {
        Article result = null;
        try {
            getById(article.getId());
            deleteById(article.getId());
            repository.add(article);
            return article;
        } catch (Exception ex) {
            repository.add(article);
            return article;
        }
    }

    @Override
    public void deleteById(Long id) {
        Article articleFound = getById(id);
        repository.remove(articleFound);
    }

    @Override
    public Article getById(Long id) {
        return repository.stream().filter(a -> id.equals(a.getId())).findFirst().orElseThrow(() -> new
RuntimeException(String.format("No article with id=%s", id)));
    }
}
```

f. La couche présentation

- Créer le package *ma.formations.soap.presentation* et ensuite créer la classe **ArticleSoapController** suivante :

```java
package ma.formations.soap.presentation;

import jakarta.annotation.PostConstruct;
import jakarta.jws.WebMethod;
import jakarta.jws.WebParam;
import jakarta.jws.WebResult;
import jakarta.jws.WebService;
import ma.formations.soap.service.IService;
import ma.formations.soap.service.ServiceImpl;
import ma.formations.soap.service.model.Article;

import java.util.List;

@WebService(serviceName = "EcommerceWS")
public class ArticleSoapController {
   private IService service;

   @WebMethod
   @WebResult(name = "Article")
   public Article saveArticle(@WebParam(name = "article") Article article) {
      return service.save(article);
   }

   @PostConstruct
   private void init() {
      this.service = new ServiceImpl();
   }

   @WebMethod
   @WebResult(name = "Article")
   public List<Article> getAll() {
      return service.getAll();
   }

   @WebMethod
   public String deleteById(Long id) {
      service.deleteById(id);
      return String.format("Article with id=%s is removed with success",id);
   }

   @WebMethod
   @WebResult(name = "Article")
   public Article getById(Long id) {
      return service.getById(id);
   }
}
```

Explications :

- Le WS doit être annoté avec **@WebService** de l'api JAX-WS. Le nom du WS qui sera généré au niveau du fichier WSDL est EcommerceWS.
- Chaque méthode du WS doit être **public** et annotée avec **@WebMethod**.
- **@WebResult** permet de personnaliser le nom de la balise qui représente le résultat de la méthode dans le fichier WSDL. Par défaut, le nom utilisé par JAX-WS est **return**.
- Eviter de surcharger les méthodes.
- **@PostConstruct** (c'est une annotation standard faisant partie de la JDK) est utilisée afin que la JVM exécute en premier la méthode init() une fois une instance de la classe ServiecImpl est créée.

e. La classe Main

- Modifier la méthode *main* de la classe **Main** comme suit :

```
package ma.formations.soap;

import jakarta.xml.ws.Endpoint;
import ma.formations.soap.presentation.ArticleSoapController;

public class Main {
   public static void main(String[] args) {
      Endpoint.publish("http://localhost:9999/ecommerce", new ArticleSoapController());
      System.out.println("Serveur démarré : http://localhost:9999/ecommerce");
   }
}
```

Explications :

- JAX-WS permet de démarrer un serveur dans le port 9999 (vous pouvez changer ce port) et publier le WS grâce à la méthode ***publish*** de la classe ***Endpoint***.

V. Tests

a. Le fichier WSDL

- Lancer la méthode main.
- Vérifier que le fichier WSDL est accessible via le lien suivant : http://localhost:9999/ecommerce?wsdl

Explications :

- Vous constatez que JAX-WS crée un schéma XML pour définir l'ensemble des types de données utilisées dans la requête et la réponse SOAP :

```
<!--  Published by XML-WS Runtime (https://github.com/eclipse-ee4j/metro-jax-ws). Runtime's version is XML-WS Runtime 4.0.2 git-revision#0264419.  -->
<!--  Generated by XML-WS Runtime (https://github.com/eclipse-ee4j/metro-jax-ws). Runtime's version is XML-WS Runtime 4.0.2 git-revision#0264419.  -->
<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
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://presentation.soap.formations.ma/" xmlns:xsd="http:/
xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://presentation.soap.formations.ma/" name="EcommerceWS">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://presentation.soap.formations.ma/" schemaLocation="http://localhost:9999/ecommerce?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="getAll">
    <part name="parameters" element="tns:getAll"/>
  </message>
  <message name="getAllResponse">
    <part name="parameters" element="tns:getAllResponse"/>
  </message>
  <message name="saveArticle">
    <part name="parameters" element="tns:saveArticle"/>
  </message>
```

dans le style document, les types contenus dans la requête et la réponse SOAP sont définis dans un schéma XML.

- Le schéma XML définit l'ensemble de types :

```
<!--  Published by XML-WS Runtime (https://github.com/eclipse-ee4j/metro-jax-ws). Runtime's version is XML-WS Runtime 4.0.2 git-revision#0264419.  -->
<xs:schema xmlns:tns="http://presentation.soap.formations.ma/" xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0" targetNamespace="http://presentation.soap.formations.ma/">
  <xs:element name="deleteById" type="tns:deleteById"/>
  <xs:element name="deleteByIdResponse" type="tns:deleteByIdResponse"/>
  <xs:element name="getAll" type="tns:getAll"/>
  <xs:element name="getAllResponse" type="tns:getAllResponse"/>
  <xs:element name="getById" type="tns:getById"/>
  <xs:element name="getByIdResponse" type="tns:getByIdResponse"/>
  <xs:element name="saveArticle" type="tns:saveArticle"/>
  <xs:element name="saveArticleResponse" type="tns:saveArticleResponse"/>
  <xs:complexType name="saveArticle">
    <xs:sequence>
      <xs:element name="article" type="tns:article" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="article">
    <xs:sequence>
      <xs:element name="description" type="xs:string" minOccurs="0"/>
      <xs:element name="id" type="xs:long" minOccurs="0"/>
      <xs:element name="price" type="xs:double" minOccurs="0"/>
      <xs:element name="quantity" type="xs:double" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

saveArticle corresonds au nom de la balise qui sera envoyée dans la requête SOAP pour sauvegarder un article.

saveArticle est une sequence composé d'un seul élément de type article.

Un article est une séquence composée de : description, id, price et quantity.

- La requête et la réponse SOAP pour sauvegarder un article :
-

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:pres="http://presentation.soap.formations.ma/">
   <soapenv:Header/>
   <soapenv:Body>
      <pres:saveArticle>
         <!--Optional:-->
         <article>
            <!--Optional:-->
            <description>Article_6</description>
            <!--Optional:-->
            <id>6</id>
            <!--Optional:-->
            <price>15000</price>
            <!--Optional:-->
            <quantity>45</quantity>
         </article>
      </pres:saveArticle>
   </soapenv:Body>
</soapenv:Envelope>
```

XML Raw

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:saveArticleResponse xmlns:ns2="http://presentation.soap.formations.ma/">
         <Article>
            <description>Article_6</description>
            <id>6</id>
            <price>15000.0</price>
            <quantity>45.0</quantity>
         </Article>
      </ns2:saveArticleResponse>
   </S:Body>
</S:Envelope>
```

- Le *portType* correspond à l'interface de votre service web (ici c'est l'interface *IService*).
- L'opération (*operation* balise) correspond au nom de la méthode définie dans l'interface du WS :

```
</message>
▼<portType name="ArticleSoapController">   C'est l'interface du WS.
 ▼<operation name="getAll">
    <input wsam:Action="http://presentation.soap.formations.ma/ArticleSoapController/getAllRequest" message="tns:getAll"/>
    <output wsam:Action="http://presentation.soap.formations.ma/ArticleSoapController/getAllResponse" message="tns:getAllResponse"/>
  </operation>
 ▼<operation name="saveArticle">
    <input wsam:Action="http://presentation.soap.formations.ma/ArticleSoapController/saveArticleRequest" message="tns:saveArticle"/>
    <output wsam:Action="http://presentation.soap.formations.ma/ArticleSoapController/saveArticleResponse" message="tns:saveArticleResponse"/>
  </operation>
 ▼<operation name="deleteById">
    <input wsam:Action="http://presentation.soap.formations.ma/ArticleSoapController/deleteByIdRequest" message="tns:deleteById"/>
    <output wsam:Action="http://presentation.soap.formations.ma/ArticleSoapController/deleteByIdResponse" message="tns:deleteByIdResponse"/>
  </operation>
 ▼<operation name="getById">
    <input wsam:Action="http://presentation.soap.formations.ma/ArticleSoapController/getByIdRequest" message="tns:getById"/>
    <output wsam:Action="http://presentation.soap.formations.ma/ArticleSoapController/getByIdResponse" message="tns:getByIdResponse"/>
  </operation>
</portType>
```

L'opération corresponds au nom de la méthode annoté par @WebMethod

- Le binding sert à lier le WS au portType (l'interface du WS).
- Au niveau du binding, on précise le style des messages SOAP (**Document** ou **RPC**), le type d'encodage des données (**Encoded** ou bien **Literal**) et le protocole de transport (ici c'est http).
- Par défaut, le style est **Document** et le type d'encodage est **Literal** :

```
▼<binding name="ArticleSoapControllerPortBinding" type="tns:ArticleSoapController">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  ▼<operation name="getAll">
     <soap:operation soapAction=""/>
    ▼<input>
       <soap:body use="literal"/>
     </input>
    ▼<output>
       <soap:body use="literal"/>
     </output>
   </operation>
  ▶<operation name="saveArticle">
   ...
   </operation>
```

c'est le portType (l'interface du WS)

L'encodage des données (Literal ou Encoded)

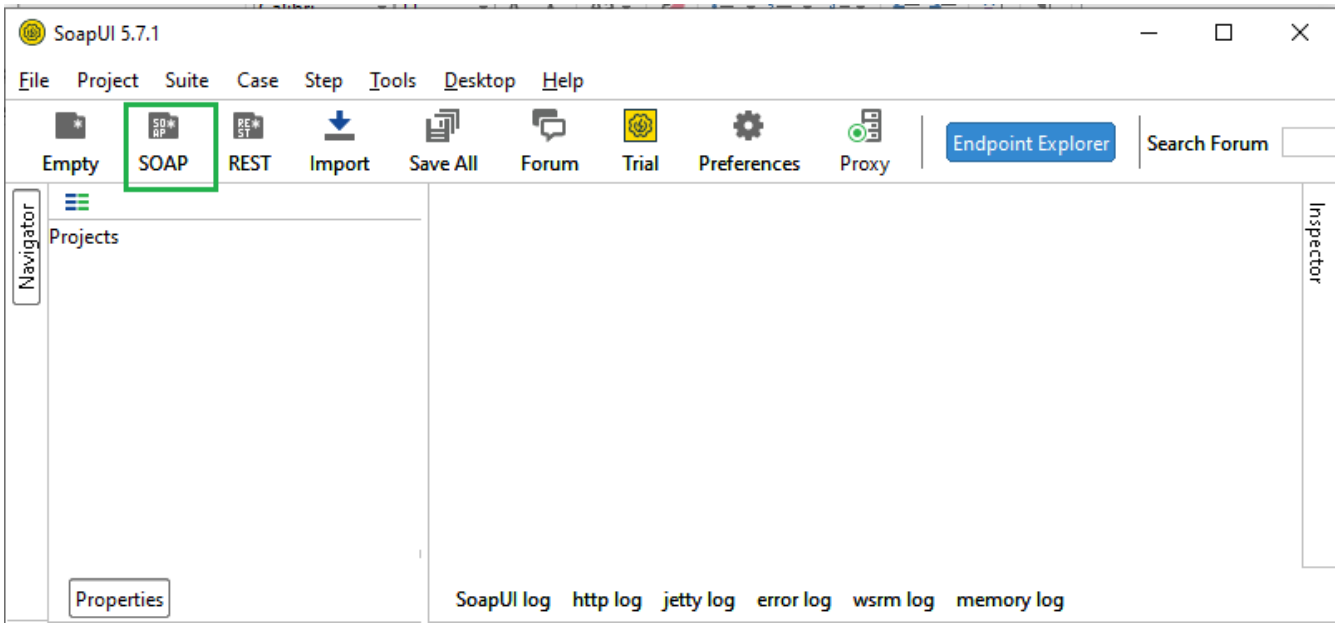
Le style des messages SOAP (Document ou RPC)

- Le service précise le nom du WS, le type du binding et l'URL du WS :

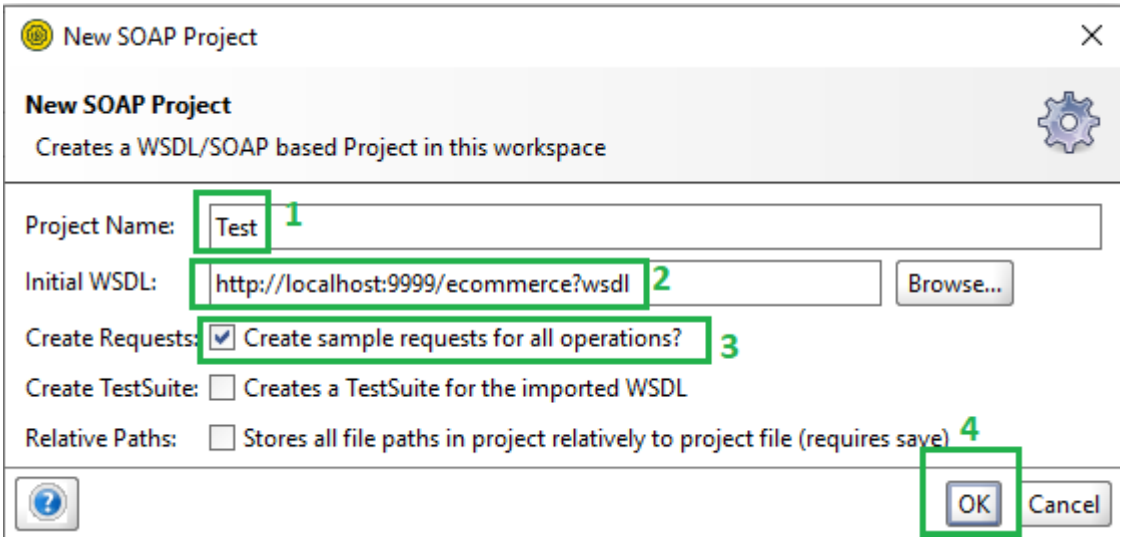
```
▼<service name="EcommerceWS">
  ▼<port name="ArticleSoapControllerPort" binding="tns:ArticleSoapControllerPortBinding">
     <soap:address location="http://localhost:9999/ecommerce"/>
   </port>
 </service>
```

b. Tester avec SOAP UI

- Lancer *SoapUI*, l'interface suivante s'affiche :



- Cliquer sur SOAP comme expliqué ci-dessus, l'écran suivant s'affiche :

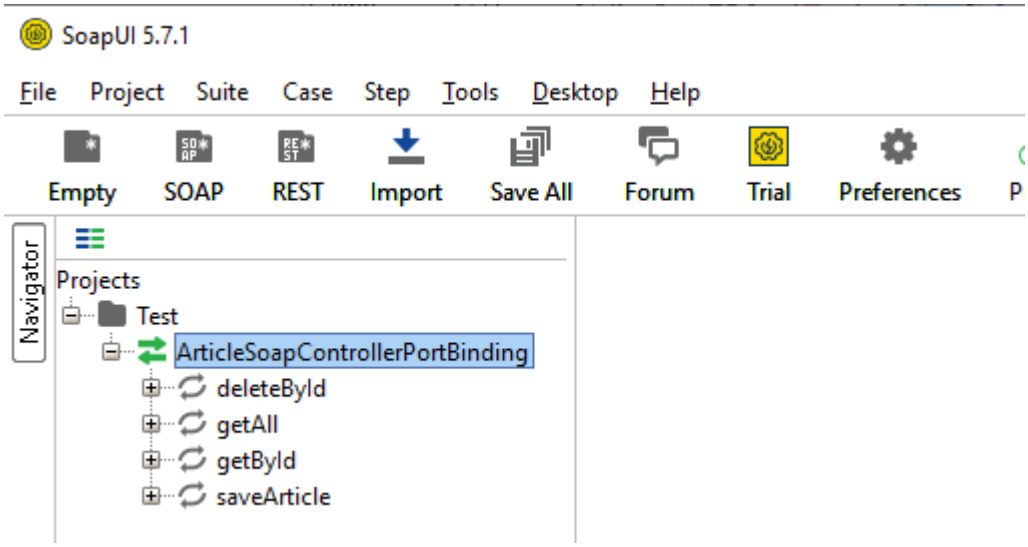


1- Entrer le nom de votre projet.

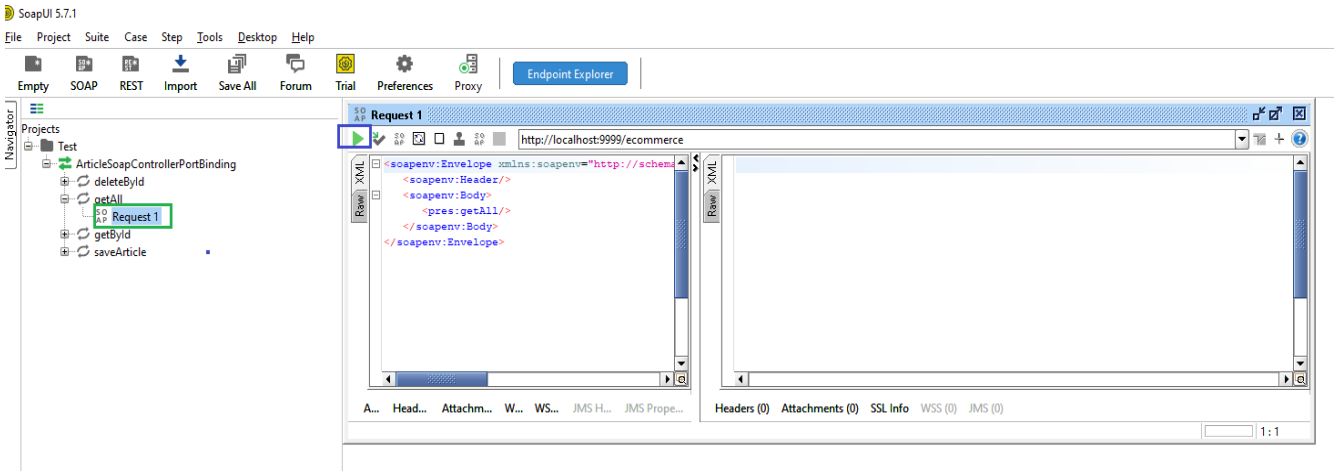
2- Entrer le lien de fichier WSDL.

3- Cocher « Create Requests ».

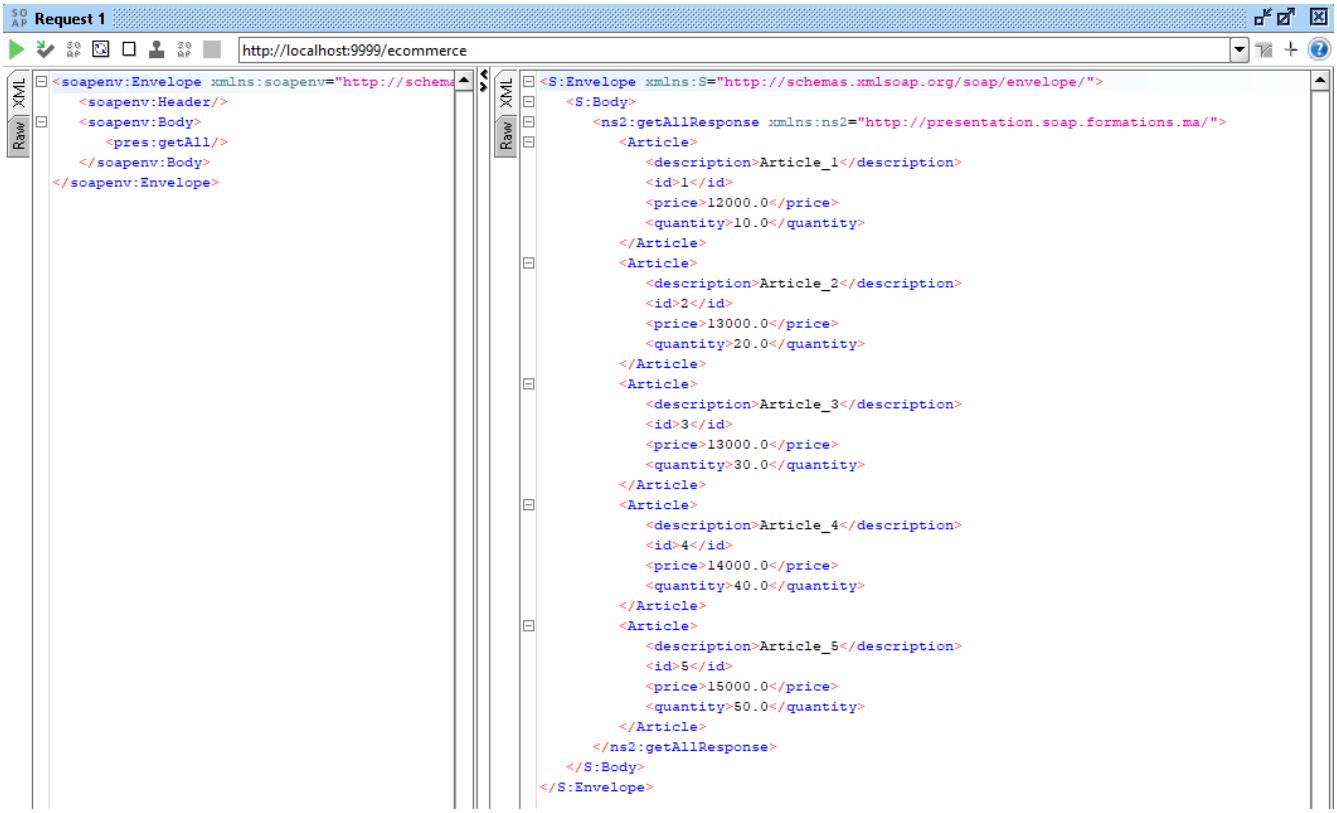
4- Cliquer sur le bouton OK. L'écran suivant s'affiche :



- Pour tester getAll, cliquer dessus ensuite cliquer sur *Request* comme expliqué ci-après :



- Cliquer sur la flèche comme expliqué ci-dessus pour exécuter la requête :



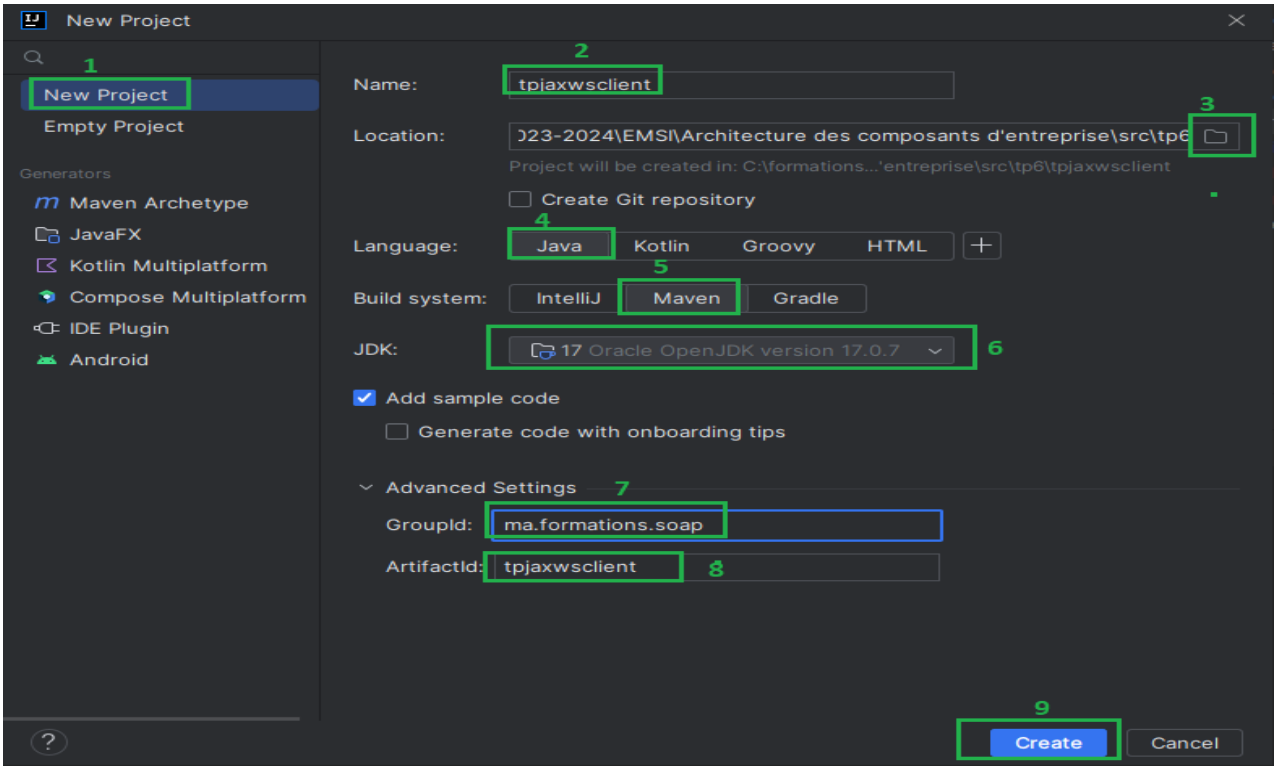
VI. Partie 2 : Développement du client avec Intellig (la version Ultimate)

Dans cette seconde partie, nous allons voir comment générer le stub en utilisant le plugin ***« Jakarta EE : Web Services (JAX-WS) »*** de la version Ultimate d'Intellig.

NB : Si vous ne disposez pas de la version Ultimate vous pouvez aller directement à la partie 3 de cet atelier.

a. Création du projet Maven

- Créer un nouveau projet Maven comme expliqué ci-après :



1. Cliquer sur *New Project*.
2. Entrer le nom de votre projet (par exemple : tpjaxwsclient)
3. Préciser le chemin de votre projet (par exemple : c:\workspace)
4. Cliquer sur *Java*.
5. Cliquer sur *Maven*.
6. Préciser JDK 17.
7. Entrer le *GroupId* (par exemple : ma.formations.soap).
8. Entrer *l'ArtifactId* (par exemple : tpjaxwsclient).
9. Cliquer sur le bouton *Create*.

b. Le fichier pom.xml

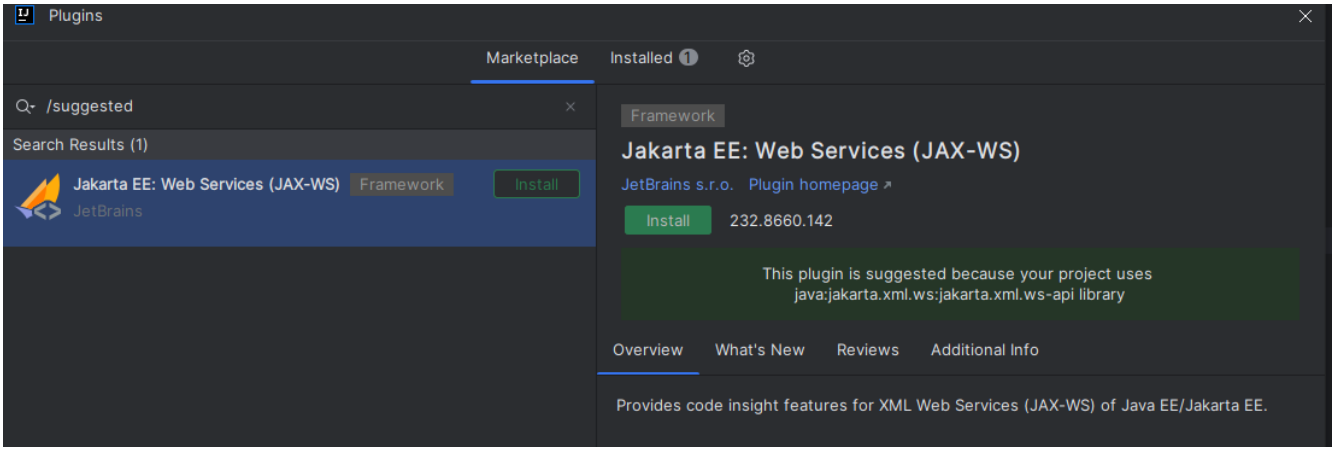
- Ajouter les dépendances suivantes au niveau du fichier pom.xml :

```
<dependency>
  <groupId>com.sun.xml.ws</groupId>
  <artifactId>jaxws-ri</artifactId>
  <version>4.0.2</version>
  <type>pom</type>
</dependency>
<dependency>
```

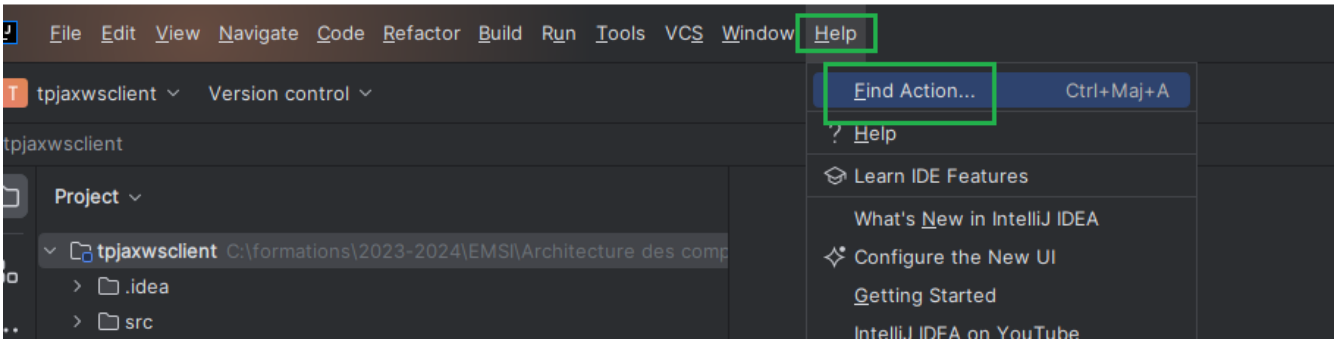
```
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.30</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.7.2</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>5.7.2</version>
  <scope>test</scope>
</dependency>
```

c. Installer le plugin Jakarta EE : Web Services (JAX-WS)

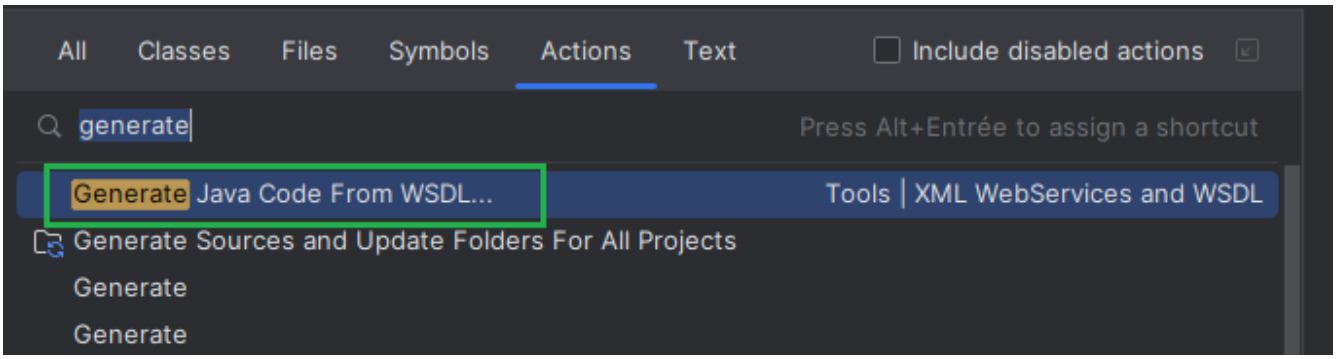
- Pour générer le Stub, commencer par installer le plugin ***« Jakarta EE : Web Services (JAX-WS)*** » au niveau d'Intellig :



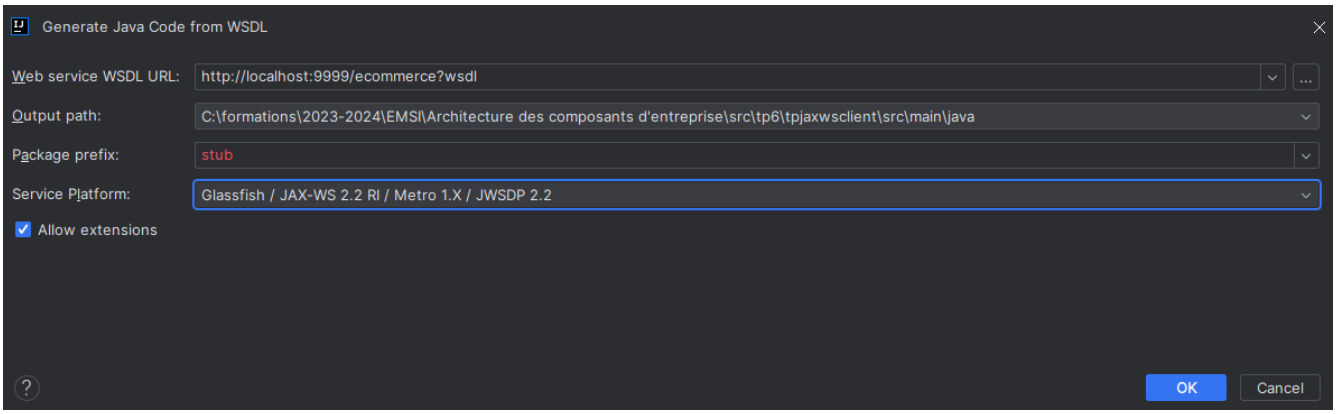
- Cliquer ensuite sur « ***Help - > Find Action…*** » comme expliqué ci-dessous :



- Cliquer ensuite sur « ***Generate Java Code From WSDL…*** » :



- Entrer le lien du WSDL (vérifier que votre WS est bien démarré), le chemin dans lequel les classes du Stub seront générés, le nom du package dans lequel seront mises les classes du Stub et l'implémentation de JAX WS.



- Cliquer ensuite sur le bouton OK. Vérifier que le plugin a bien généré les classes du Stub.
- Annoter la classe Article qui a été généré par le plugin avec les annotations suivantes : @Data @Builder, @AllArgsConstructor, @NoArgsConstructor@Data de Lombok, nous en aurons besoin pour le développement des cas de tests.

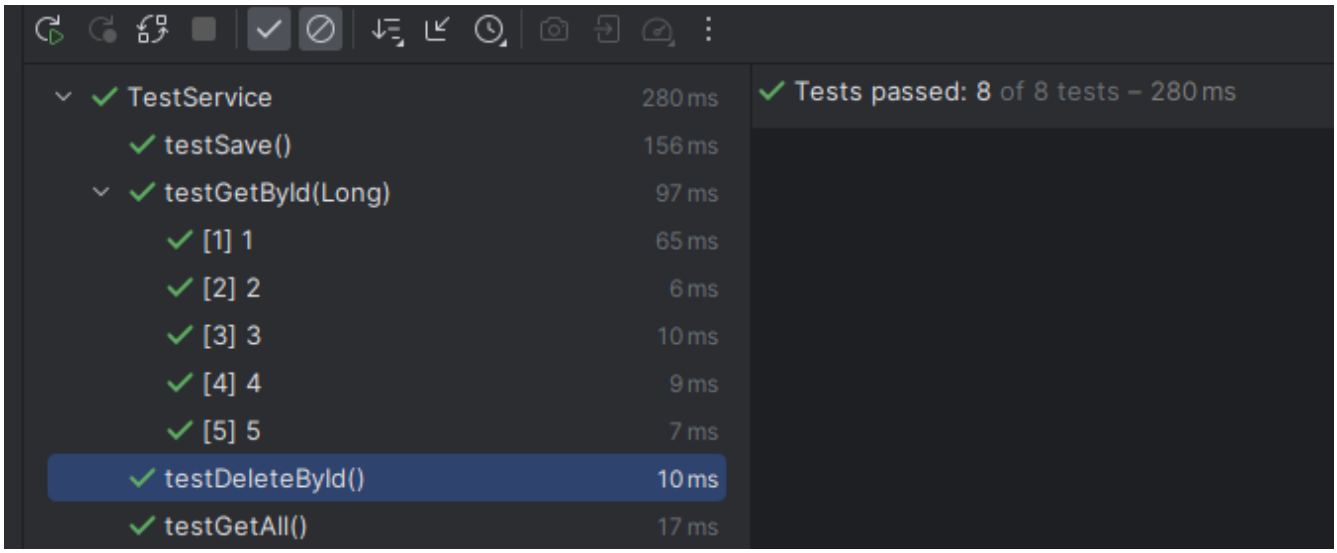
d. Créer la classe de test avec JUNIT

- Créer la classe **TestWS** suivante :

```
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import stub.Article;
import stub.ArticleSoapController;
import stub.EcommerceWS;

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

import static org.junit.jupiter.api.Assertions.assertAll;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class TestWS {
    private static ArticleSoapController proxy = null;
    private static final List<Article> expectedData = new ArrayList<>();

    @BeforeAll
    public static void init() {
        proxy = new EcommerceWS().getArticleSoapControllerPort();
        expectedData.add(Article.builder().id(1l).description("Article_1").price(12000.0).quantity(10.0).build());
        expectedData.add(Article.builder().id(2l).description("Article_2").price(13000.0).quantity(20.0).build());
        expectedData.add(Article.builder().id(3l).description("Article_3").price(13000.0).quantity(30.0).build());
        expectedData.add(Article.builder().id(4l).description("Article_4").price(14000.0).quantity(40.0).build());
        expectedData.add(Article.builder().id(5l).description("Article_5").price(15000.0).quantity(50.0).build());
    }

    @Test
    public void testGetAll() {

        List<Article> result = proxy.getAll();
        assertEquals(5, result.size());
    }

    @ParameterizedTest
    @ValueSource(longs = {1, 2, 3, 4, 5})
    public void testGetById(Long id) {
        Article expectedArticleById=expectedData.stream().filter(a->a.getId().equals(id)).findFirst().get();
        Article result = proxy.getById(id);
        assertAll(
                () -> assertEquals(expectedArticleById.getId(), result.getId()),
                () -> assertEquals(expectedArticleById.getDescription(), result.getDescription()),
                () -> assertEquals(expectedArticleById.getPrice(), result.getPrice()),
                () -> assertEquals(expectedArticleById.getQuantity(), result.getQuantity())
        );
    }

    @Test
    public void testDeleteById() {
```

```
    String result = proxy.deleteById(1l);
    assertEquals(String.format("Article with id=%s is removed with success", 1l), result);
  }

  @Test
  public void testSave() {
    Article article=Article.builder().id(6l).description("Article_6").price(32000.0).quantity(55.0).build();
    proxy.saveArticle(article);

    Article result = proxy.getById(6l);
    assertAll(
        () -> assertEquals(article.getId(), result.getId()),
        () -> assertEquals(article.getDescription(), result.getDescription()),
        () -> assertEquals(article.getPrice(), result.getPrice()),
        () -> assertEquals(article.getQuantity(), result.getQuantity())
    );
  }
}
```

- Lancer les tests et vérifier le résultat suivant :

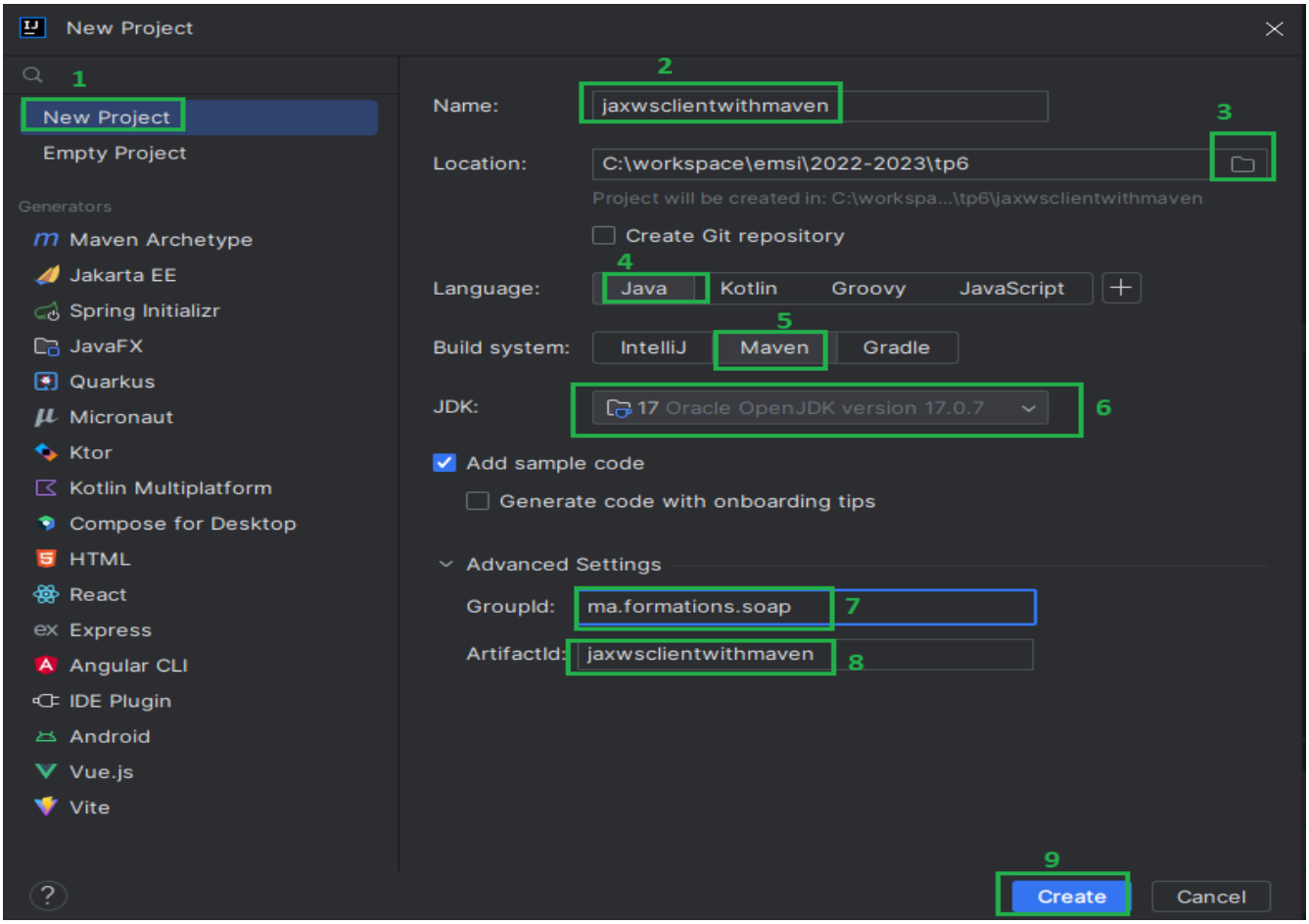


VII. Partie 3 : Développement du client avec Maven

Dans cette troisième partie, nous allons voir comment générer le stub en utilisant le plugin ***jaxws-maven-plugin*** de Maven (développé par MojoHaus : https://www.mojohaus.org/).

a. Création du projet Maven

- Créer un nouveau projet Maven comme expliqué ci-après :



1. Cliquer sur *New Project*.
2. Entrer le nom de votre projet (par exemple : tpjaxwsclientwithmaven)
3. Préciser le chemin de votre projet (par exemple : c:\workspace)
4. Cliquer sur *Java*.
5. Cliquer sur *Maven*.
6. Préciser JDK 17.
7. Entrer le *GroupId* (par exemple : ma.formations.soap).
8. Entrer *l'ArtifactId* (par exemple : tpjaxwsclientwithmaven).

9. Cliquer sur le bouton *Create*.

b. Le fichier pom.xml

- Ajouter les dépendances suivantes au niveau du fichier pom.xml :

```
<dependency>
  <groupId>com.sun.xml.ws</groupId>
  <artifactId>jaxws-ri</artifactId>
  <version>4.0.2</version>
  <type>pom</type>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.30</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.7.2</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>5.7.2</version>
  <scope>test</scope>
</dependency>
```

Explications :

- ✓ La dépendance **jaxws-ri** est nécessaire pour pouvoir compiler le projet une fois le stub est généré par le plugin.
- ✓ La dépendance **junit-jupiter-params** est nécessaire pour développer les tests paramétrés. Il s'agit en fait des tests unitaires dont la même méthode peut être exécutée plusieurs fois selon les valeurs qui lui seront passés en paramètre.

- Ajouter le plugin suivant au niveau du fichier pom.xml :

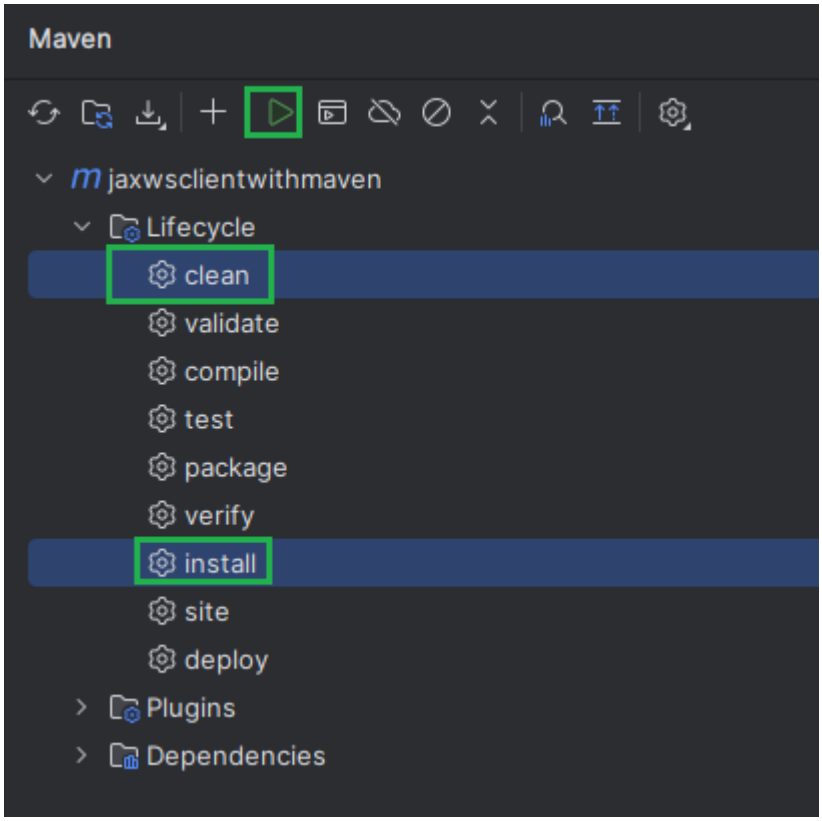
```
<build>
  <plugins>
    <plugin>
      <groupId>com.sun.xml.ws</groupId>
      <artifactId>jaxws-maven-plugin</artifactId>
      <version>4.0.1</version>
      <executions>
        <execution>
          <goals>
            <goal>wsimport</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <wsdlUrls>
          <wsdlUrl>http://localhost:9999/ecommerce?wsdl</wsdlUrl>
        </wsdlUrls>
        <packageName>stub</packageName>
        <sourceDestDir>
          ${project.basedir}/src/main/java
        </sourceDestDir>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Explications :

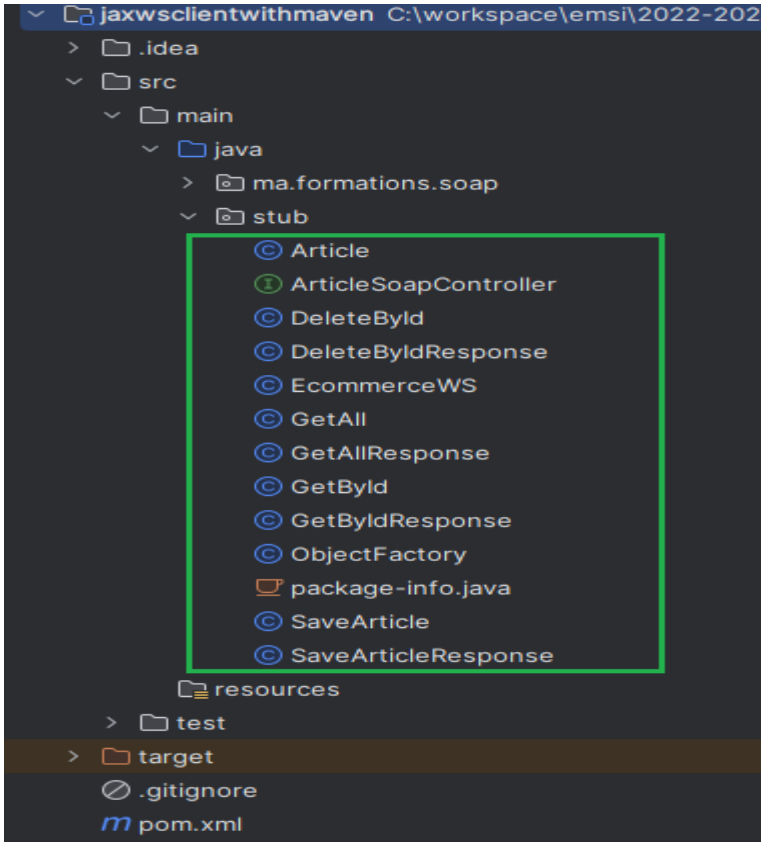
- Dans la balise ***<wsdlUrl>***, préciser l'URL de votre WS (ce dernier doit être démarré pour pouvoir l'invoquer par la suite).
- Dans la balise ***<packageName>,*** préciser le nom du package dans lequel les classes du stub seront créées par le plugin.
- Dans la balise ***<sourceDestDir>***, préciser le chemin du dossier dans lequel les classes du stub seront créées par le plugin.
- Une fois vous lancez la commande ***« mvn clean install »***, le plugin analysera le fichier WSDL et générera les classes du Stub dans le package stub au niveau du dossier */src/main/java*. La génération du Stub se fait moyennant la commande ***wsimport*** de JAX WS (***wsimport -keep http://localhost:9999/ecommerce?wsdl***).

c. Exécuter le plugin

- Au niveau d'Intellig, lancer la commande ***mvn clean install*** comme expliqué ci-dessous :



- Vérifier que le plugin a bien généré les classes du Stub comme montré ci-après :



- Annoter la classe Article qui a été généré par le plugin avec les annotations suivantes : @Data @Builder, @AllArgsConstructor, @NoArgsConstructor@Data de Lombok, nous en aurons besoin pour le développement des cas de tests.

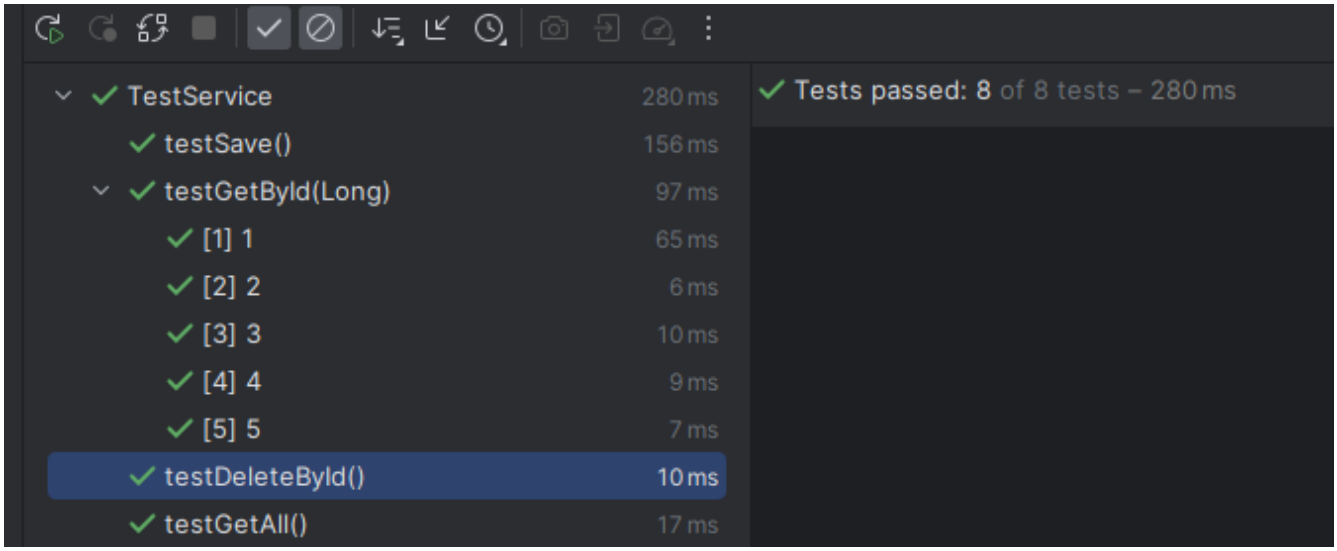
d. Créer la classe de test avec JUNIT

- Créer la classe **TestWS** suivante :

```
package ma.formations.soap;

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import stub.Article;
import stub.ArticleSoapController;
import stub.EcommerceWS;

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

import static org.junit.jupiter.api.Assertions.assertAll;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class TestWS {

    private static ArticleSoapController proxy = null;
    private static final List<Article> expectedData = new ArrayList<>();

    @BeforeAll
    public static void init() {
        proxy = new EcommerceWS().getArticleSoapControllerPort();

expectedData.add(Article.builder().id(1l).description("Article_1").price(12000.0).quantity(10.0).build());

expectedData.add(Article.builder().id(2l).description("Article_2").price(13000.0).quantity(20.0).build());

expectedData.add(Article.builder().id(3l).description("Article_3").price(13000.0).quantity(30.0).build());

expectedData.add(Article.builder().id(4l).description("Article_4").price(14000.0).quantity(40.0).build());

expectedData.add(Article.builder().id(5l).description("Article_5").price(15000.0).quantity(50.0).build());
    }

    @Test
    public void testGetAll() {

        List<Article> result = proxy.getAll();
        assertEquals(5, result.size());
```

```
    }

    @ParameterizedTest
    @ValueSource(longs = {1, 2, 3, 4, 5})
    public void testGetById(Long id) {
        Article expectedArticleById=expectedData.stream().filter(a->a.getId().equals(id)).findFirst().get();
        Article result = proxy.getById(id);
        assertAll(
                () -> assertEquals(expectedArticleById.getId(), result.getId()),
                () -> assertEquals(expectedArticleById.getDescription(), result.getDescription()),
                () -> assertEquals(expectedArticleById.getPrice(), result.getPrice()),
                () -> assertEquals(expectedArticleById.getQuantity(), result.getQuantity())
        );
    }

    @Test
    public void testDeleteById() {
        String result = proxy.deleteById(1l);
        assertEquals(String.format("Article with id=%s is removed with success", 1l), result);
    }

    @Test
    public void testSave() {
        Article article=Article.builder().id(6l).description("Article_6").price(32000.0).quantity(55.0).build();
        proxy.saveArticle(article);

        Article result = proxy.getById(6l);
        assertAll(
                () -> assertEquals(article.getId(), result.getId()),
                () -> assertEquals(article.getDescription(), result.getDescription()),
                () -> assertEquals(article.getPrice(), result.getPrice()),
                () -> assertEquals(article.getQuantity(), result.getQuantity())
        );
    }
}
```

- Lancer les tests et vérifier le résultat suivant :



Conclusion

- Le code source de cet atelier est disponible sur GITHUB :

  Pour la partie serveur : https://github.com/abbouformations/jaxws.git

  Pour la partie client avec le plugin d'Intellig : https://github.com/abbouformations/jaxwsclient.git

  Pour la partie client avec Maven : https://github.com/abbouformations/jaxwsclientwithmaven.git