

9.12 Activité pratique : Web Service SOAP avec JAX-WS et Spring Boot

Objectif du TP

Ce TP a pour objectif de :

- Mettre en place un service web SOAP avec Spring Boot et Apache CXF.
- Configurer les dépendances nécessaires pour un projet Spring Boot avec support SOAP.
- Gérer des entités JPA dans un contexte de service web.

Prérequis

Préparer :

- Un environnement de développement Java (Java 20 recommandé).
- Maven pour la gestion des dépendances.
- Une connaissance de base en Spring Boot et en services SOAP.

La structure finale du projet est la suivante :

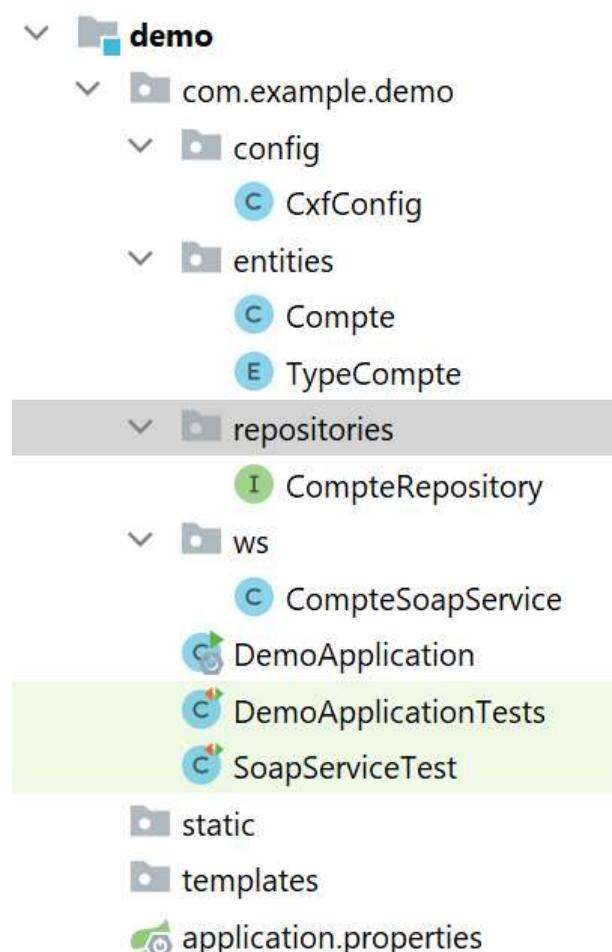


Figure 9.11: Structure finale du projet.

Étape 1 : Initialisation du Projet Spring Boot

Créer un nouveau projet Spring Boot en utilisant Spring Initializr en ligne ou un IDE compatible. Choisir Maven comme système de gestion de projet, Java comme langage, et inclure les dépendances suivantes :

- **Spring Web** : pour les fonctionnalités REST et le support de serveur web.
- **Spring Data JPA** : pour la gestion des entités et l'intégration de JPA.
- **H2 Database** : pour une base de données en mémoire utilisée pendant le développement.

- **Lombok** : pour simplifier le code en générant automatiquement les getters, setters et autres méthodes via des annotations.
- **Spring Boot DevTools** : pour activer le redémarrage automatique du serveur pendant le développement, ce qui améliore la productivité.

Étape 2 : Ajout des Dépendances Apache CXF pour SOAP

Pour intégrer le support de SOAP dans le projet Spring Boot, il est nécessaire d'ajouter des dépendances spécifiques pour Apache CXF. Apache CXF est une bibliothèque qui facilite la création de services web SOAP et RESTful. En ajoutant ces dépendances, il devient possible de publier et consommer des services web SOAP dans une application Spring Boot.

Ajouter les dépendances suivantes dans le fichier pom.xml du projet :

```
<dependencies>
    <!-- Apache CXF Core pour le support SOAP -->
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-core</artifactId>
        <version>4.0.2</version>
    </dependency>

    <!-- Apache CXF Starter pour JAX-WS avec Spring Boot -->
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-spring-boot-starter-jaxws</artifactId>
        <version>4.0.2</version>
    </dependency>
</dependencies>
```

- **cxf-core** : Cette dépendance inclut les composants principaux d'Apache CXF nécessaires pour implémenter les services SOAP. Elle fournit les classes et interfaces de base pour la création de services web SOAP et RESTful.
- **cxf-spring-boot-starter-jaxws** : Cette dépendance simplifie l'intégration de CXF avec Spring Boot, en ajoutant un support pour JAX-WS (Java API for XML Web Services). Elle permet de définir et publier des services web SOAP dans l'application Spring Boot. La version spécifiée (4.0.2) est compatible avec Jakarta EE et garantit le support des dernières fonctionnalités SOAP.



Ces dépendances sont indispensables pour créer et configurer un service web SOAP avec Apache CXF dans un projet Spring Boot. Assurez-vous de bien les inclure dans le fichier pom.xml avant de passer à la configuration des services.

Étape 3 : Configuration de la Base de Données

La base de données H2 est utilisée dans ce projet pour stocker les données de manière temporaire pendant le développement. H2 est une base de données en mémoire légère et intégrée, idéale pour les tests et le développement rapide. La configuration de H2 permet de vérifier que le service SOAP fonctionne correctement avec une base de données avant de passer à une base de données de production, si nécessaire.

Configuration dans application.properties

Pour configurer la base de données H2 dans Spring Boot, ajouter les paramètres suivants dans le fichier src/main/resources/application.properties :

```
# URL de la base de données H2 en mémoire
```

```

spring.datasource.url=jdbc:h2:mem:testdb

# Nom d'utilisateur de la base de données
spring.datasource.username=sa

# Mot de passe de la base de données (vide pour H2)
spring.datasource.password=

# Driver de la base de données H2
spring.datasource.driverClassName=org.h2.Driver

# Configuration de JPA pour la création automatique des tables
spring.jpa.hibernate.ddl-auto=update

# Activer la console H2 pour visualiser les données pendant le développement
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

```

Explication des Paramètres

- **spring.datasource.url=jdbc:h2:mem:testdb** : Spécifie l'URL de la base de données H2. Le préfixe jdbc:h2:mem indique que la base de données sera créée en mémoire, avec testdb comme nom.
- **spring.datasource.username=sa** et **spring.datasource.password=** : Définissent les identifiants de connexion pour accéder à la base de données. Par défaut, H2 utilise le nom d'utilisateur sa et un mot de passe vide.
- **spring.datasource.driverClassName=org.h2.Driver** : Indique le driver JDBC à utiliser pour H2.
- **spring.jpa.hibernate.ddl-auto=update** : Ce paramètre de JPA Hibernate permet de créer ou mettre à jour automatiquement les tables en fonction des entités définies. Dans ce cas, les tables correspondant aux entités comme Compte seront générées automatiquement.
- **spring.h2.console.enabled=true** et **spring.h2.console.path=/h2-console** : Ces paramètres activent la console H2, accessible depuis <http://localhost:8082/h2-console>. La console permet de visualiser les données stockées dans la base, exécuter des requêtes SQL, et gérer les tables pendant le développement.

Accès à la Console H2

Pour vérifier les données stockées dans la base H2 :

1. Lancer l'application Spring Boot.
2. Dans un navigateur, accéder à <http://localhost:8082/h2-console>.
3. Dans la page de connexion de la console H2, renseigner les informations suivantes :
 - **JDBC URL** : jdbc:h2:mem:testdb
 - **User Name** : sa
 - **Password** : laisser vide
4. Cliquer sur Connect pour accéder aux tables de la base de données et visualiser les données des comptes.



La configuration de H2 en mode en mémoire permet de supprimer toutes les données une fois l'application arrêtée, ce qui est pratique pour le développement. Cependant, pour des environnements de production, il est recommandé de configurer une base de données permanente comme MySQL ou PostgreSQL.

Étape 4 : Création de l'Entité Compte

Définir l'entité Compte pour représenter un compte bancaire. Annoter la classe avec `@Entity` pour la gestion de persistance via JPA. Utiliser également des annotations JAXB (Java Architecture for XML Binding) pour permettre la sérialisation XML lors de la communication SOAP. Ces annotations JAXB facilitent la conversion de l'objet Java en XML, ce qui est essentiel pour les services web SOAP, où les données sont échangées au format XML.

```

package com.example.demo.entities;

import jakarta.persistence.*;
import jakarta.xml.bind.annotation.XmlAccessType;
import jakarta.xml.bind.annotation.XmlAccessorType;
import jakarta.xml.bind.annotation.XmlRootElement;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Compte {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private double solde;

    @Temporal(TemporalType.DATE)
    private Date dateCreation;

    @Enumerated(EnumType.STRING)
    private TypeCompte type;
}

```

- **`@XmlRootElement`** : Cette annotation indique que la classe `Compte` peut être la racine d'un document XML. Elle est utilisée pour marquer la classe en tant qu'élément racine lors de la sérialisation en XML, ce qui permet de créer un élément XML principal pour chaque instance de `Compte`. Par exemple, si une instance de `Compte` est convertie en XML, le document XML généré commencera par un élément `<Compte>`.
- **`@XmlAccessorType(XmlAccessType.FIELD)`** : Cette annotation spécifie le niveau d'accès pour la sérialisation XML. En utilisant `XmlAccessType.FIELD`, tous les champs de la classe sont inclus dans la sérialisation XML, même s'ils ne possèdent pas de méthodes getter ou setter publiques. Cela permet de s'assurer que toutes les propriétés de l'entité sont correctement sérialisées en XML, sans nécessiter des méthodes d'accès explicites.
- **`@XmlElement`** (non explicitement utilisé ici, mais souvent employé) : Cette annotation permet de personnaliser les noms des éléments XML pour chaque champ si besoin. Elle est souvent utilisée pour donner un nom spécifique à chaque propriété dans le XML. Par exemple, pour renommer le champ `id` en `compteId` dans l'élément XML, il suffirait d'annoter `id` avec `@XmlElement(name = "compteId")`.



Les annotations JAXB sont essentielles pour les services web SOAP, car elles permettent de contrôler la façon dont les objets Java sont convertis en XML et inversement. Cela garantit une compatibilité optimale avec le format XML utilisé dans la communication SOAP. Dans cet exemple, les annotations `@XmlRootElement` et `@XmlAttributeType` facilitent la transformation de l'entité `Compte` en un format XML structuré, conforme aux spécifications des services SOAP.

Étape 5 : Création du Repository Compte

Créer un repository en étendant `JpaRepository` pour fournir des méthodes CRUD sur l'entité `Compte`.

```
package com.example.demo.repositories;

import com.example.demo.entities.Compte;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CompteRepository extends JpaRepository<Compte, Long> {
```

Étape 6 : Création du Service SOAP

Définir un service SOAP pour exposer des méthodes permettant la gestion des comptes bancaires. Ce service permet de récupérer la liste des comptes, d'obtenir un compte par ID, de créer un nouveau compte, et de supprimer un compte existant.

Annoter la classe et les méthodes avec des annotations spécifiques à SOAP pour les rendre accessibles via ce protocole.

```
package com.example.demo.ws;

import com.example.demo.entities.Compte;
import com.example.demo.entities.TypeCompte;
import com.example.demo.repositories.CompteRepository;

import jakarta.jws.WebMethod;
import jakarta.jws.WebParam;
import jakarta.jws.WebService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.List;
import java.util.Optional;

@Component
@WebService(serviceName = "BanqueWS")
public class CompteSoapService {

    @Autowired
    private CompteRepository compteRepository;

    @WebMethod
    public List<Compte> getComptes() {
        return compteRepository.findAll();
```

```

    }

    @WebMethod
    public Compte getCompteById(@WebParam(name = "id") Long id) {
        return compteRepository.findById(id).orElse(null);
    }

    @WebMethod
    public Compte createCompte(@WebParam(name = "solde") double solde,
                               @WebParam(name = "type") TypeCompte type) {
        Compte compte = new Compte(null, solde, new Date(), type);
        return compteRepository.save(compte);
    }

    @WebMethod
    public boolean deleteCompte(@WebParam(name = "id") Long id) {
        if (compteRepository.existsById(id)) {
            compteRepository.deleteById(id);
            return true;
        }
        return false;
    }
}

```

- **@WebService** : Cette annotation, placée au niveau de la classe CompteSoapService, indique que cette classe représente un service web SOAP. Le paramètre serviceName définit le nom du service (ici, "BanqueWS"). Ce nom sera utilisé dans le WSDL (Web Services Description Language) pour identifier le service.
- **@Component** : Annotation de Spring qui enregistre la classe en tant que bean dans le contexte de l'application Spring. Cela permet à Spring d'injecter automatiquement les dépendances, comme le CompteRepository dans ce cas.
- **@Autowired** : Indique à Spring d'injecter automatiquement le CompteRepository, qui est le composant permettant d'interagir avec la base de données. Grâce à cette injection, il est possible d'utiliser les méthodes du repository pour gérer les opérations CRUD sur les comptes.
- **@WebMethod** : Annotation qui marque une méthode comme étant accessible via le service web SOAP. Toutes les méthodes annotées avec @WebMethod deviennent des opérations du service SOAP et peuvent être appelées par des clients SOAP.
- **@WebParam** : Annotation qui permet de renommer les paramètres d'entrée dans le message SOAP. Dans chaque méthode, cette annotation renomme les paramètres pour qu'ils apparaissent clairement dans la requête SOAP. Par exemple, dans la méthode getCompteById, le paramètre id est annoté avec @WebParam(name = "id"), ce qui spécifie que ce paramètre doit être identifié comme id dans le message SOAP.

Détails des Méthodes Exposées

- **getComptes()** : Cette méthode retourne la liste de tous les comptes disponibles dans la base de données. Grâce à @WebMethod, elle est exposée comme une opération SOAP qui peut être invoquée pour récupérer tous les comptes.
- **getCompteById(Long id)** : Retourne un compte spécifique en fonction de son identifiant. Le paramètre id est annoté avec @WebParam pour être clairement identifiable dans la requête SOAP. Si l'identifiant ne correspond à aucun compte, la méthode retourne null.
- **createCompte(double solde, TypeCompte type)** : Crée un nouveau compte bancaire avec un solde initial et un type de compte (par exemple, COURANT ou EPARGNE). La méthode utilise le constructeur de l'entité Compte pour initialiser l'objet avec une date de création (définie

automatiquement à la date actuelle), puis sauvegarde cet objet en base de données.

- **deleteCompte(Long id)** : Supprime un compte en fonction de son identifiant. Avant de supprimer le compte, la méthode vérifie s'il existe dans la base de données via `compteRepository.existsById(id)`. Si le compte existe, il est supprimé et la méthode retourne `true`. Sinon, elle retourne `false` pour indiquer l'absence de compte correspondant à cet identifiant.

R

Les annotations `@WebService` et `@WebMethod` sont essentielles pour exposer les méthodes en tant que services SOAP. Elles assurent une compatibilité avec les spécifications SOAP et permettent aux clients de consommer ces services facilement. En utilisant Spring et Apache CXF, la configuration et la publication des services SOAP sont simplifiées, ce qui facilite le développement d'applications basées sur des services web.

Étape 7 : Configuration d'Apache CXF

Pour rendre le service SOAP accessible aux clients, il est nécessaire de configurer un endpoint (point d'accès) pour le service en définissant un bean dans une classe de configuration dédiée. Cette classe, appelée `CxfConfig`, lie le service SOAP au chemin `/ws`, permettant aux clients d'accéder au service via une URL bien définie.

R

La classe `CxfConfig` utilise Apache CXF pour publier le service SOAP `CompteSoapService`. En définissant un endpoint via CXF, le service est rendu accessible à travers le protocole SOAP.

```
package com.example.demo.config;

import com.example.demo.ws.CompteSoapService;
import lombok.AllArgsConstructor;
import org.apache.cxf.Bus;
import org.apache.cxf.jaxws.EndpointImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@AllArgsConstructor
public class CxfConfig {

    private CompteSoapService compteSoapService;
    private Bus bus;

    @Bean
    public EndpointImpl endpoint() {
        EndpointImpl endpoint = new EndpointImpl(bus, compteSoapService);
        endpoint.publish("/ws");
        return endpoint;
    }
}
```

Explication des composants de la classe `CxfConfig`

- **@Configuration** : Cette annotation indique à Spring que la classe `CxfConfig` est une classe de configuration. Spring traitera cette classe comme une source de définitions de beans, ce qui signifie que les méthodes annotées avec `@Bean` seront exécutées pour créer et enregistrer des beans dans le contexte de l'application.

- **@AllArgsConstructor** : Annotation de Lombok qui génère automatiquement un constructeur prenant en paramètre tous les attributs de la classe. Ici, elle permet d'injecter les dépendances CompteSoapService et Bus directement dans la classe via le constructeur.
- **private CompteSoapService compteSoapService** : L'instance de CompteSoapService représente le service SOAP défini précédemment. Cette instance sera exposée en tant que service SOAP accessible depuis l'extérieur de l'application. Elle contient toutes les méthodes annotées @WebMethod qui seront publiées via CXF.
- **private Bus bus** : Le Bus est un composant central d'Apache CXF. Il gère la configuration et les extensions pour CXF, y compris les services SOAP. En passant ce bus à l'endpoint, CXF peut gérer les fonctionnalités nécessaires pour la communication SOAP.
- **@Bean public EndpointImpl endpoint()** : Cette méthode crée et configure l'endpoint du service SOAP. L'annotation @Bean enregistre cette méthode en tant que bean Spring, ce qui permet de l'initialiser automatiquement.
- **EndpointImpl endpoint = new EndpointImpl(bus, compteSoapService)** : La classe EndpointImpl de CXF est utilisée pour configurer et publier un endpoint pour le service SOAP. Le constructeur prend le bus CXF et le service SOAP (ici, compteSoapService) comme paramètres, établissant ainsi une liaison entre le service et le bus CXF.
- **endpoint.publish("/ws")** : Cette ligne publie le service SOAP sur le chemin spécifié, ici /ws. En publiant ce service à ce chemin, le service SOAP sera accessible à l'URL <http://localhost:8080/ws> (en supposant que le serveur est en cours d'exécution sur le port 8080). Les clients pourront alors interagir avec ce service en envoyant des requêtes SOAP à cette URL.



La configuration de CXF via un `EndpointImpl` est essentielle pour exposer le service SOAP. Elle fournit un point d'accès stable et structuré pour les clients SOAP qui souhaitent interagir avec le service. Grâce à cette configuration, l'application Spring Boot est capable de gérer la communication SOAP en exploitant pleinement les fonctionnalités de CXF.

Étape 8 : Tester le Service SOAP avec SoapUI

SoapUI est un outil populaire pour tester les services web SOAP. Cette étape explique comment récupérer le fichier WSDL du service, puis utiliser SoapUI pour envoyer des requêtes et tester les différentes méthodes exposées par le service SOAP.

Récupération du WSDL

Le WSDL (Web Services Description Language) est un document XML qui décrit les services web, les opérations qu'ils exposent, et les paramètres qu'ils attendent. Dans ce projet, le WSDL est généré automatiquement par Apache CXF et peut être récupéré depuis l'URL suivante (en supposant que le serveur est en cours d'exécution sur le port 8082) :

- URL du WSDL : <http://localhost:8082/services/ws?wsdl>

Configuration de SoapUI pour Tester le Service

1. **Lancer SoapUI** : Ouvrir l'application SoapUI.
2. **Créer un Nouveau Projet SOAP** : Dans SoapUI, sélectionner File > New SOAP Project pour créer un nouveau projet.
3. **Saisir le Nom du Projet** : Donner un nom au projet, comme TestBanqueSOAP.
4. **Fournir l'URL du WSDL** : Dans le champ Initial WSDL, entrer l'URL du WSDL : <http://localhost:8082/services/ws?wsdl>.
5. **Valider la Création du Projet** : Cliquer sur OK. SoapUI va automatiquement analyser le WSDL et générer les requêtes SOAP pour chaque méthode exposée par le service.

Tester les Méthodes du Service SOAP

Une fois le projet créé, SoapUI génère des requêtes pour chaque opération disponible dans le service. Voici comment tester les principales méthodes :

1. Tester la Méthode getComptes :

- Dans l'arborescence du projet SoapUI, sélectionner la requête getComptes.
- Cliquer sur l'onglet Request 1 pour ouvrir l'éditeur de requêtes.
- Cliquer sur Submit pour envoyer la requête. La réponse contiendra la liste des comptes bancaires disponibles.

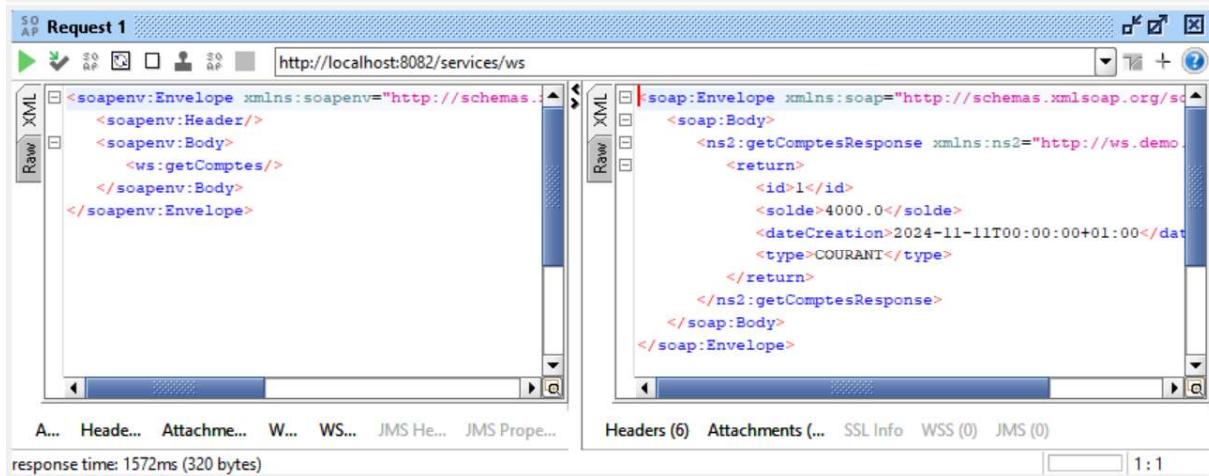


Figure 9.12: Test dans SoapUI.

2. Tester la Méthode getCompteById :

- Sélectionner la requête getCompteById.
- Dans le message SOAP généré, renseigner l'ID du compte dans le champ id.
- Cliquer sur Submit pour envoyer la requête. La réponse contiendra les détails du compte correspondant à l'ID spécifié.

3. Tester la Méthode createCompte :

- Sélectionner la requête createCompte.
- Remplir les paramètres requis dans le message SOAP :
 - solde : saisir le solde initial du compte.
 - type : spécifier le type de compte (COURANT ou EPARGNE).
- Cliquer sur Submit. La réponse contiendra les détails du compte nouvellement créé.

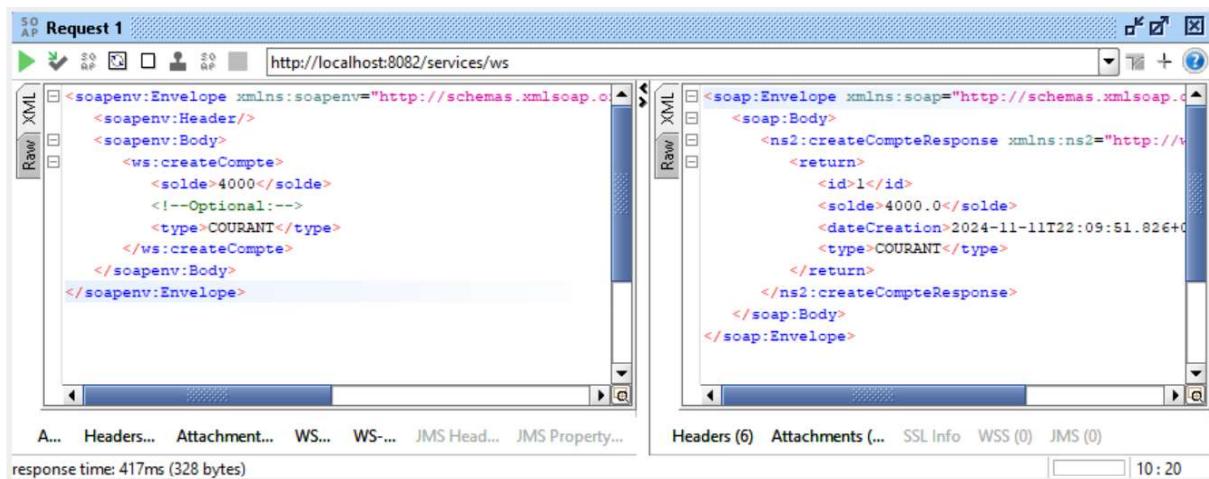


Figure 9.13: Test dans SoapUI.

4. Tester la Méthode deleteCompte :

- Sélectionner la requête deleteCompte.

- Renseigner l'ID du compte à supprimer dans le champ `id`.
- Cliquer sur `Submit`. La réponse indiquera si l'opération a été réalisée avec succès (vrai) ou si l'ID était introuvable (faux).

En utilisant SoapUI, il est possible de tester facilement les différentes méthodes du service SOAP. Le WSDL fournit toutes les informations nécessaires pour que SoapUI génère les requêtes et les réponses associées, ce qui permet de s'assurer que le service fonctionne correctement et que chaque méthode répond comme prévu.