| Lab 2: SOA/SOAP with Java (Spring-WS) | A.Y.: 2025/2026 |
| --- | --- |
| Course: SoA and Microservices<br>Class: 4Info<br>Instructor: Dr. Salah Gontara | |

# Lab Objectives

- Understand the contract-first approach in SOAP (XSD → WSDL → stubs/objects).
- Deploy and run a SOAP service (Spring Boot + Spring-WS).
- Test SOAP operations with Postman (requests, responses, Fault errors). • Observe the impact of the contract (XSD) on exchanged messages.

# Prerequisites

- Java JDK 17+
- Maven 3.8+
- Postman (desktop)
- Git

You must fork the following GitLab repository, then work on your fork:
https://gitlab.com/tps-soa-microservices/soap-bank

# Functional context

The SOAP service simulates a minimal banking system with accounts (e.g., A100, B200) and provides typical operations:

- GetAccount: view an account's information (owner, balance, currency).
- Deposit: deposit an amount and obtain the new balance.

# Required work

## A) Fork, clone, and run

1. Fork the soap-bank repository to your GitLab account.

2. Clone your fork locally.

3. Build and run the application.

4. Check that the WSDL is accessible via a web browser.

```
C:\Users\asus1\OneDrive\Bureau\fac\2emeCycle\java>git clone https://github.com/Manarfekih/BANK_SOAP.git
Cloning into 'BANK_SOAP'...
remote: Enumerating objects: 74, done.
remote: Counting objects: 100% (74/74), done.
remote: Compressing objects: 100% (44/44), done.
remote: Total 74 (delta 8), reused 74 (delta 8), pack-reused 0 (from 0)
Receiving objects: 100% (74/74), 21.83 KiB | 254.00 KiB/s, done.
Resolving deltas: 100% (8/8), done.
```

```
C:\Users\asus1\OneDrive\Bureau\fac\2emeCycle\java>cd BANK_SOAP

C:\Users\asus1\OneDrive\Bureau\fac\2emeCycle\java\BANK_SOAP>
```

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:sch="http://example.com/bank"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://example.com/bank"
 targetNamespace="http://example.com/bank">
  ▼<wsdl:types>
    ▼<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
     targetNamespace="http://example.com/bank">
      ▼<xsd:complexType name="AccountType">
        ▼<xsd:sequence>
            <xsd:element name="accountId" type="xsd:string"/>
            <xsd:element name="owner" type="xsd:string"/>
            <xsd:element name="balance" type="xsd:decimal"/>
            <xsd:element name="currency" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      ▼<xsd:element name="GetAccountRequest">
        ▼<xsd:complexType>
          ▼<xsd:sequence>
              <xsd:element name="accountId" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      ▼<xsd:element name="GetAccountResponse">
        ▼<xsd:complexType>
          ▼<xsd:sequence>
              <xsd:element name="account" type="tns:AccountType"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      ▼<xsd:element name="DepositRequest">
        ▼<xsd:complexType>
          ▼<xsd:sequence>
              <xsd:element name="accountId" type="xsd:string"/>
              <xsd:element name="amount" type="xsd:decimal"/>
            </xsd:sequence>
          </xsd:complexType>
```

## B) Reading the contract

1. Identify the XSD file and explain its role.

we have xsd file located in src/main/resources/xsd/bank.xsd

it defines:

- Data types
- Request structure
- Response structure

- Validation rules

  —-> t is the **formal contract** between client and server

2. List the request/response elements (names, fields, types).

**GetAccountResponse**

- accountId (string)
- owner (string)
- balance (decimal)
- currency (string)

**DepositRequest**

- accountId (string)
- amount (decimal)

**DepositResponse**

- accountId
- newBalance

3. From the WSDL, locate:

• the namespace,

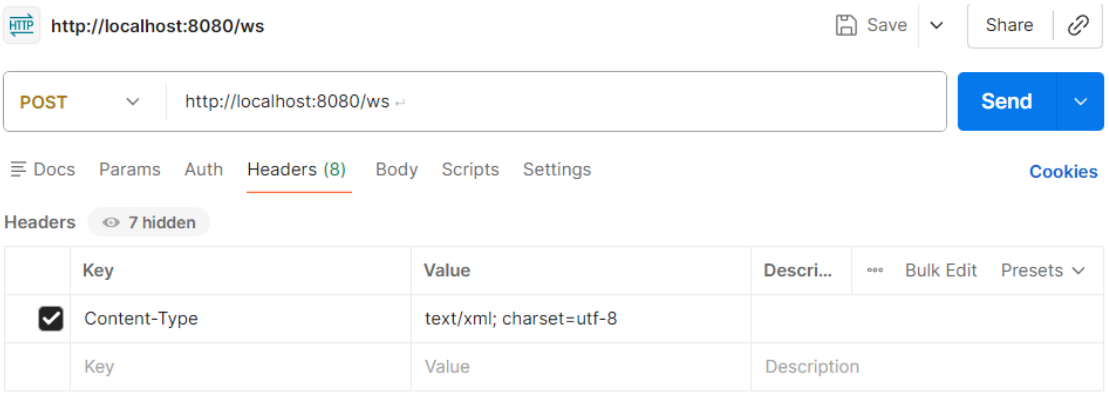

• the portType and operations,

• the endpoint (URL) and the SOAP binding.

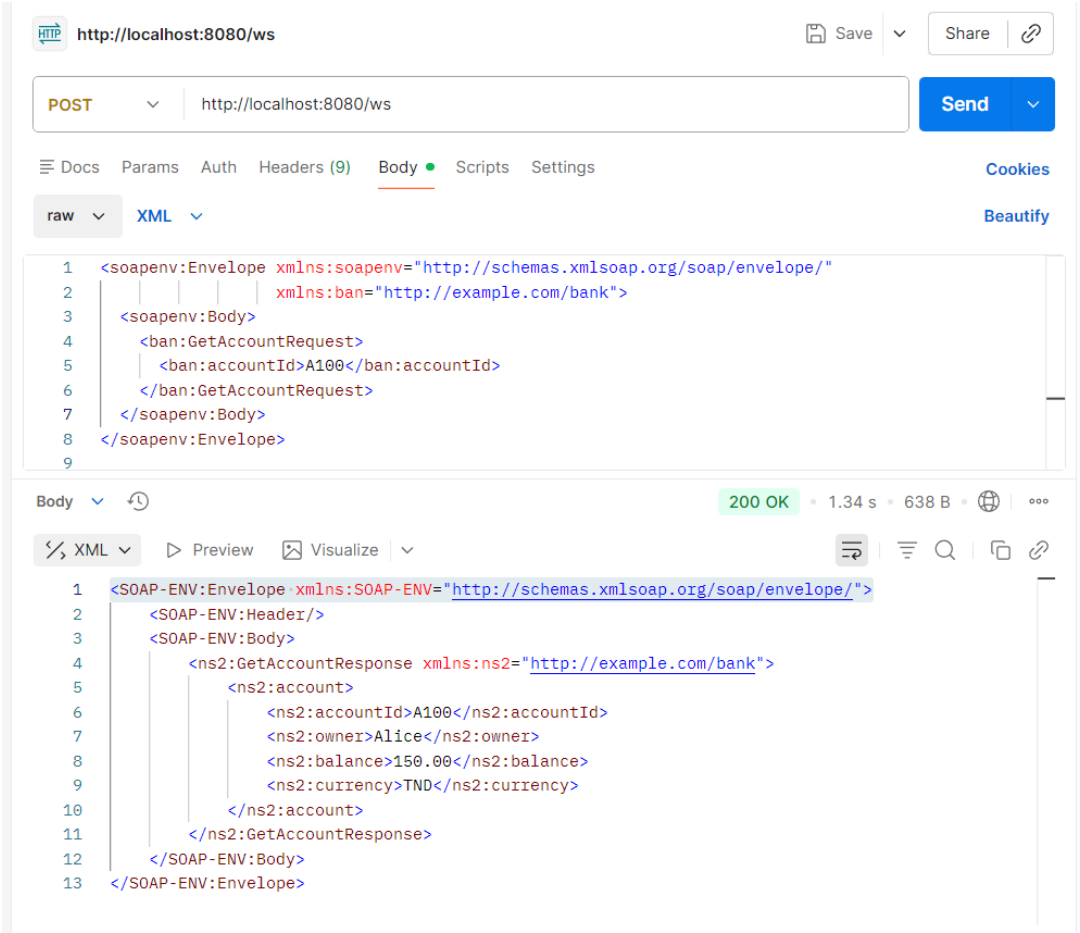

## C) Postman tests (SOAP)

1. Create a Postman request of type POST to the SOAP endpoint.

2. Add the header Content-Type: text/xml; charset=utf-8.




3. Send a GetAccount request for account A100 and analyze the response.




4. Send a Deposit request on A100 (e.g., 20.00) and verify the new balance.

Docs Params Auth Headers (9) Body Scripts Settings

raw XML

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:ban="http://example.com/bank">
  <soapenv:Body>
    <ban:DepositRequest>
      <ban:accountId>A100</ban:accountId>
      <ban:amount>20.00</ban:amount>
    </ban:DepositRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Body 200 OK • 461 ms • 516 B

XML Preview Visualize

```xml
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns2:DepositResponse xmlns:ns2="http://example.com/bank">
            <ns2:newBalance>170.00</ns2:newBalance>
        </ns2:DepositResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

5. Intentionally trigger an error (e.g., negative amount or non-existing account) and observe the SOAP Fault structure.

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:ban="http://example.com/bank">
  <soapenv:Body>
    <ban:GetAccountRequest>
      <ban:accountId>A300</ban:accountId>
    </ban:GetAccountRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Body 500 Internal Server Error • 323 ms • 523 B

XML Preview Debug with AI

```xml
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode>SOAP-ENV:Client</faultcode>
            <faultstring xml:lang="en">Unknown accountId: A300</faultstring>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

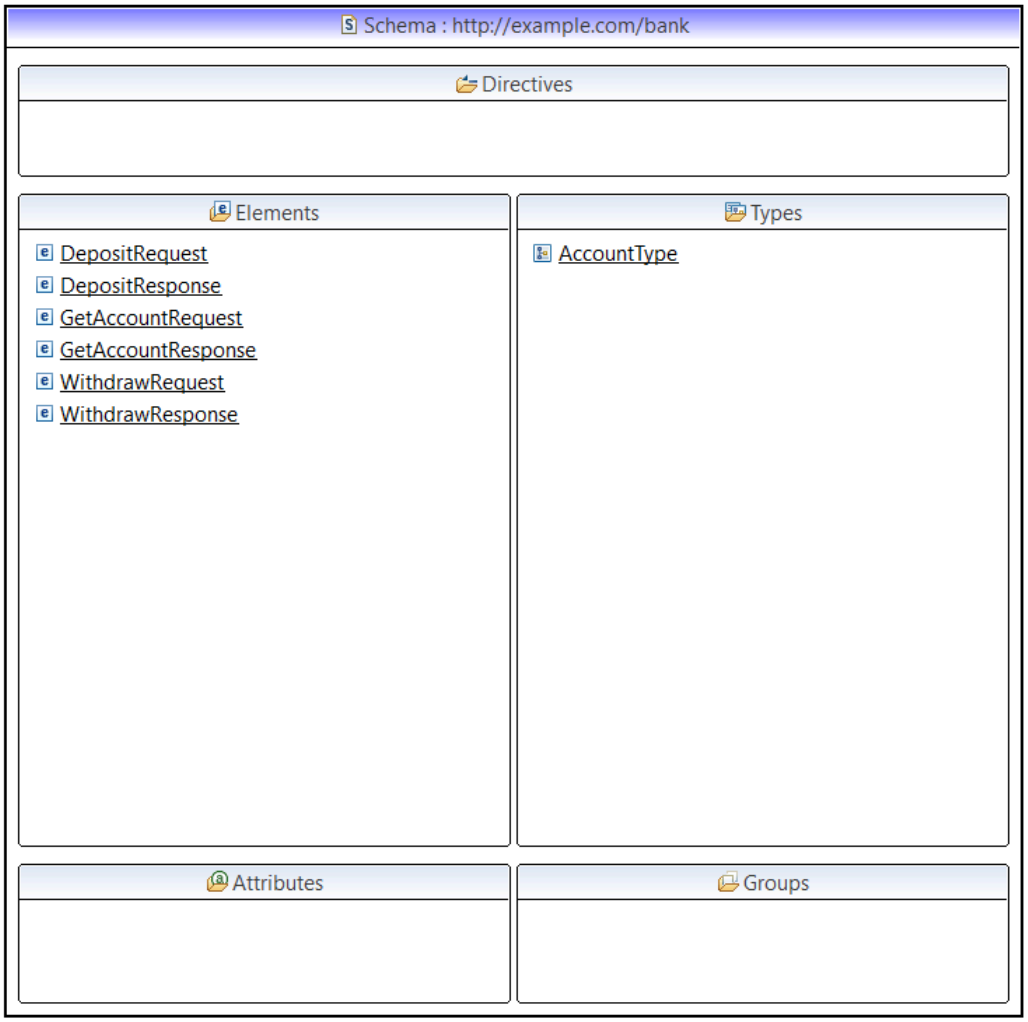
## D) Add one feature

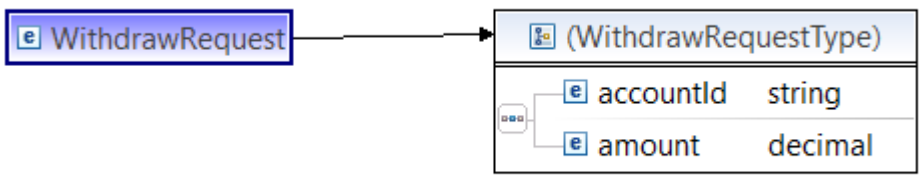
After validating the Postman tests, you must extend the SOAP service with at least one new operation, following the contract-first approach.

### 1) Choose a feature

Choose one operation to add, for example:

- Withdraw: withdraw an amount
- Transfer: transfer between two accounts
- CreateAccount: create an account



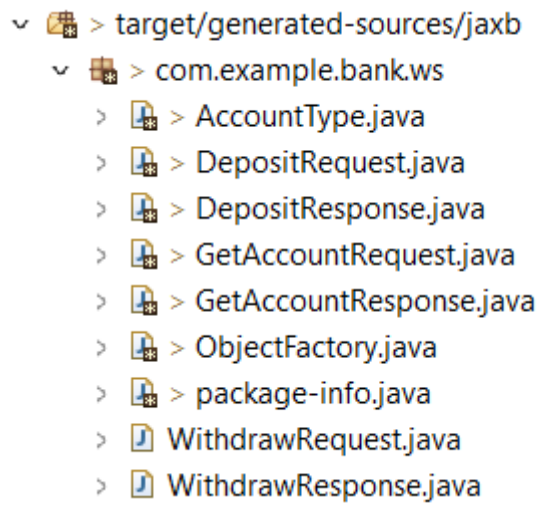



2) Update the contract

Edit the project's XSD to add the request/response elements related to the chosen operation (and the required fields).

3) Implement on the server side

Implement the operation in the Spring-WS endpoint/service using the classes generated (or updated) from the XSD.




```
  </wsdl:operation>
  <wsdl:operation name="Withdraw">
    <soap:operation soapAction=""/>
    <wsdl:input name="WithdrawRequest">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="WithdrawResponse">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="BankPortService">
  <wsdl:port binding="tns:BankPortSoap11" name="BankPortSoap11">
    <soap:address location="http://localhost:8080/ws"/>
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

4) Test with Postman

Add a Postman request for the operation:

- one nominal test (valid case)
- one error test producing a SOAP Fault (invalid amount, non-existing account, insufficient balance, etc.)

we have :

```
db.put("A100", new Account("A100", "Alice", new BigDecimal("150.00"), "TND"));
```

after test: