Exercise 3

SOAP Web Services (Java)

Author: Mariusz Fraś

1 Objectives of the exercise

The purpose of the exercise is:

1. Acquaintance with architecture of W3C SOAP Web Services.
2. Mastering the technique of creating services in Java with the use of XML Web Services API.

2 Development environment

The developing of applications can be performed using various platforms. Here is considered the following environment:

- Windows 7 (or higher) Operating System (eventually MacOS).
- Java Software Development Kit (JDK 11 or newer).
- Java application development platform (here: IntelliJ IDEA).

3 Exercise – Part I
Creating a W3C SOAP Web Service in Java

In the exercise will be implemented an application (SOAP W3C web service and client) using: XML-Based Web Services API (JAX-WS), in particular:

- Implementation of the W3C Web Service using the Bottom-Up method.
- Implementation of the W3C Web Service using the Top-Down method.
- Java client implementation using classes generated using Bottom-Up method.

The implementation of the Web Service using the Bottom-Up method consists in defining appropriate interfaces and classes for the service and then using appropriate tools to generate the target code.

The implementation of the service using the Top-Down method consists in defining the WSDL service description file (description of message formats, types, addresses, etc. defining all logical and technical aspects of the service) and on its basis generating the source code of interfaces and classes for building the application (both the service and client). Next implementing Service class with service operations code.

3.1 W3C Web Service (JAX-WS framework) - Bottom-Up method

3.1.1 Java Maven project and POM file

- Create the standard Maven project (here named jaxws1).
- Add in the POM file the section **dependences** and one **dependency** for using package **jaxws-rt** from **com.sun.xml.ws**. (containing the core Jakarta XML Web Services APIs).
  This permits to use annotations to build necessary objects.

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
<groupId>org.example</groupId>
<artifactId>jaxws1</artifactId>
<version>1.0-SNAPSHOT</version>
    ...
    <dependencies>
        <dependency>
            <groupId>com.sun.xml.ws</groupId>
            <artifactId>jaxws-rt</artifactId>
            <version>3.0.2</version>
        </dependency>
    </dependencies>
</project>
```

Note:
*Originally, API components for implementing Web services were included in Java EE in **javax.xml.ws** package (jaxws-api artifact Id) – last ver. in 2018. Next it was moved*

*to Eclipse Foundation (EE4J – Eclipse Enterprise for Java) as a part of Eclipse Metro project – **jakarta.xml.ws** package. It is recommended to use reference implementation from sun (as in this exercise **jaxws-rt** runtime of **com.sun.xml.ws** package).*

- If necessary reload maven project and download used packages (in Maven tab on the right)

3.1.2 Interfaces and classes to support operation on data

Implement the following interfaces/classes (names may be changed) :

- Person – containing same personal data and getters/setters,
- PersonRepository – interface of data repository.
- PersonRepositoryImpl – data repository,
- PersonExistsEx – exception thrown when trying add person with the same ID,
- PersonNotFoundEx - exception thrown when trying get not existent person,

- Define the class for personal data:

```
public class Person {
    private int id;
    private String firstName;
    private int age;
    public Person() {
    }
    public Person(int id, String firstName, int age) {
        this.id = id;
        this.firstName = firstName;
        this.age = age;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    [...] //the rest of getters and setters (getFirstName(), getAge(), ...
}
```

- Define classes of exceptions. Note the classes are annotated with **@WebFault annotation** (imported from jakarta.xml.ws package).

```
import jakarta.xml.ws.WebFault;
@WebFault
public class PersonExistsEx extends Exception {
    public PersonExistsEx() {
        super("This person already exists");
    }
}
```

and:

```
import jakarta.xml.ws.WebFault;
@WebFault
public class PersonNotFoundEx extends Exception {
    public PersonNotFoundEx() {
        super("The specified person does not exist");
    }
}
```

- Define the data repository interface. The interface specifies typical CRUD operations.

```
public interface PersonRepository {
  List<Person> getAllPersons();
  Person getPerson(int id) throws PersonNotFoundEx;
  Person updatePerson(int id, String name, int age) throws
                                                    PersonNotFoundEx;
  boolean deletePerson(int id) throws PersonNotFoundEx;
  Person addPerson(int id, String name, int age) throws PersonExistsEx;
  int countPersons();
}
```

Define the class implementing interface. Initialize the repository with some data.

```
public class PersonRepositoryImpl implements PersonRepository {
  private List<Person> personList;
  public PersonRepositoryImpl() {
    personList = new ArrayList<>();
    personList.add(new Person(1, "Mariusz" , 9));
    personList.add(new Person(2, "Andrzej", 10));
    ...
  }
  public List<Person> getAllPersons() {
    return personList;
  }
  public Person getPerson(int id) throws PersonNotFoundEx {
    for (Person thePerson : personList) {
      if (thePerson.getId() == id) {
        return thePerson;
      }
    }
    throw new PersonNotFoundEx();
  }
  public Person addPerson(int id, String name, int age) throws
                                                      PersonExistsEx {
    for (Person thePerson : personList) {
      if (thePerson.getId() == id) {
      throw new PersonExistsEx();
      }
    }
    Person person = new Person(id, name, age);
    personList.add(person);
    return person;
  }
```

```
    public boolean deletePerson(int id) throws PersonNotFoundEx {
      [... check if person of given id exists and remove or throw exeption ]
    }
    public Person updatePerson(int id, String name, int age) throws
                                                          PersonNotFoundEx {
      [... check if person of given id exists and update or throw exeption ]
    }
    public int countPersons() {
      return personList.size();
    }
}
```

3.1.3 Web Service

Implement the interface and class of the web Service.

- Define the interface of the service.
  Annotate the interface and methods with **@WebService** and **@WebMethod** annotations.

```
@WebService
public interface PersonService {
    @WebMethod
    Person getPerson(int id) throws PersonNotFoundEx;
    @WebMethod
    Person addPerson(int id, String name, int age) throws PersonExistsEx;
    @WebMethod
    boolean deletePerson(int id) throws PersonNotFoundEx;
    @WebMethod
    List<Person> getAllPersons();
    @WebMethod
    int countPersons();
}
```

- Define the class implementing the interface.
  - In the **@WebService** annotation include **serviceName** and **endpointInterface** attributes.
  - In the class include the initialized data repository.
  - Complete the methods with calls of repository methods and printing comments similarly as for first two methods.

```
@WebService(serviceName = "PersonService",
            endpointInterface = "org.example.jaxws.server.PersonService")
public class PersonServiceImpl implements PersonService {
    private PersonRepository dataRepository = new DataRepository();
    @WebMethod
    public Person getPerson(int id) throws PersonNotFoundEx {
        System.out.println("...called getPerson id="+id);
        return dataRepository.getPerson(id);
    }
```

```
    @WebMethod
    public List<Person> getAllPersons() {
        System.out.println("...called getAllPersons”);
        return dataRepository.getAllPersons();
    }
    @WebMethod
    public Person addPerson(int id, String name, int age) throws
                                                        PersonExistsEx {
        [...]
    }
    @WebMethod
    public boolean deletePerson(int id) throws PersonNotFoundEx {
        [...]
    }
    @WebMethod
    public int countPersons() {
        [...]
    }
}
```

3.1.4 Hosting the service (main class)

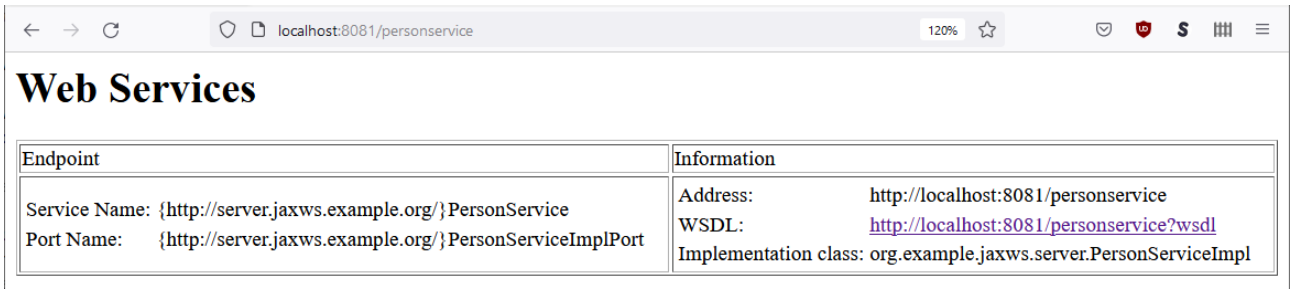
To run the service the service implementation object is created and it is “published” with use of **jakarta.xml.ws.Endpoint**.

- Define the main class of the project that hosts and run the Web Service (here named **ServiceHost**).
  - Use in the service endpoint (address) a TCP port and a name for this endpoint.

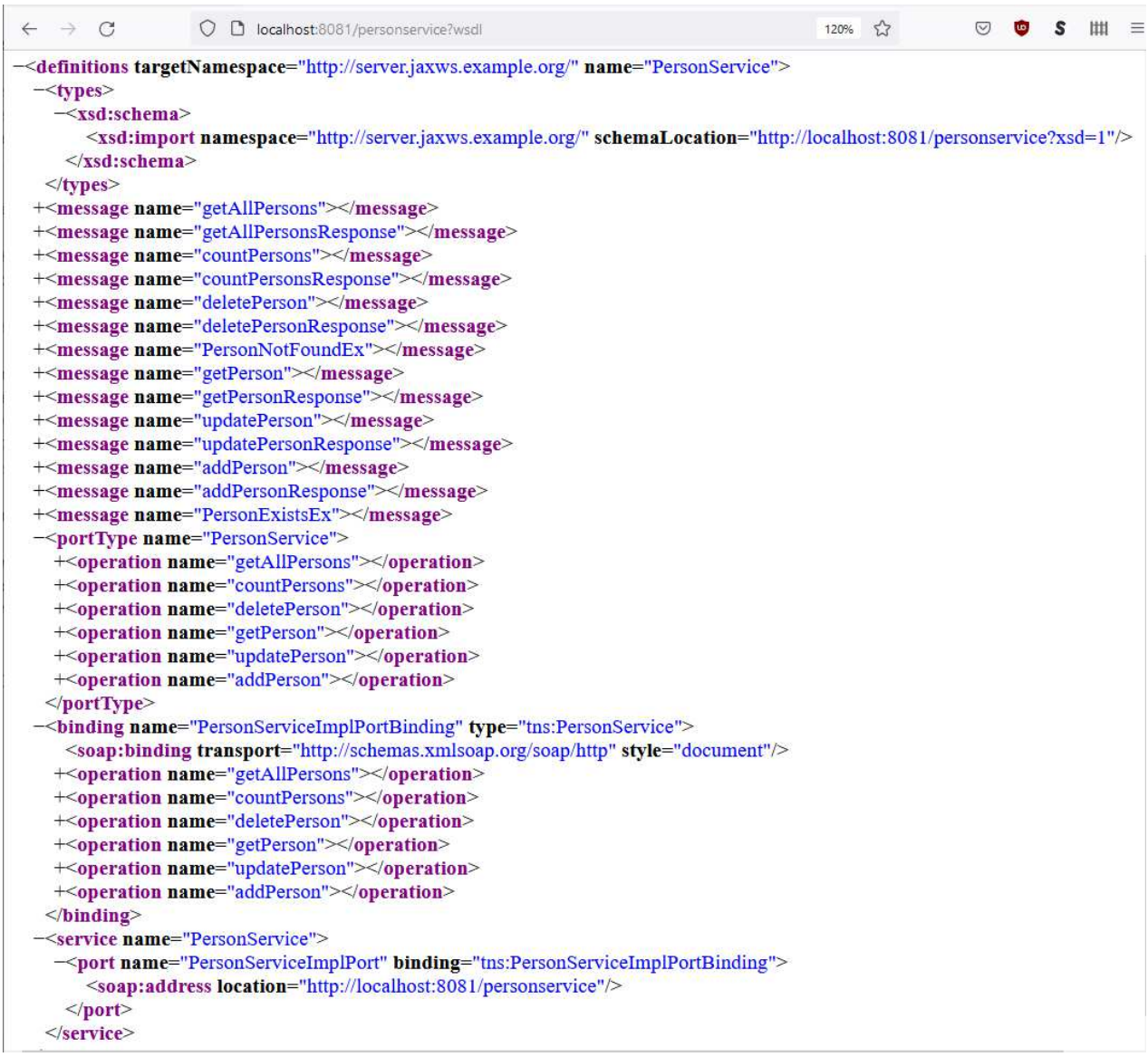
```
public class ServiceHost {
    public static void main(String[] args) {
        System.out.println("Web Service PersonService is running ...");
        PersonServiceImpl psi = new PersonServiceImpl();
        Endpoint.publish("http://localhost:8081/personservice", psi);
        System.out.println("Press ENTER to STOP PersonService ...");
        try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
        exit(0);
    }
}
```

3.1.5 Running and testing the service

- Compile and run the application
- Run the browser and enter in the address the address used for endpoint.
  - Check if in the browser the WSDL data are displayed – if yes it means the service was started properly.

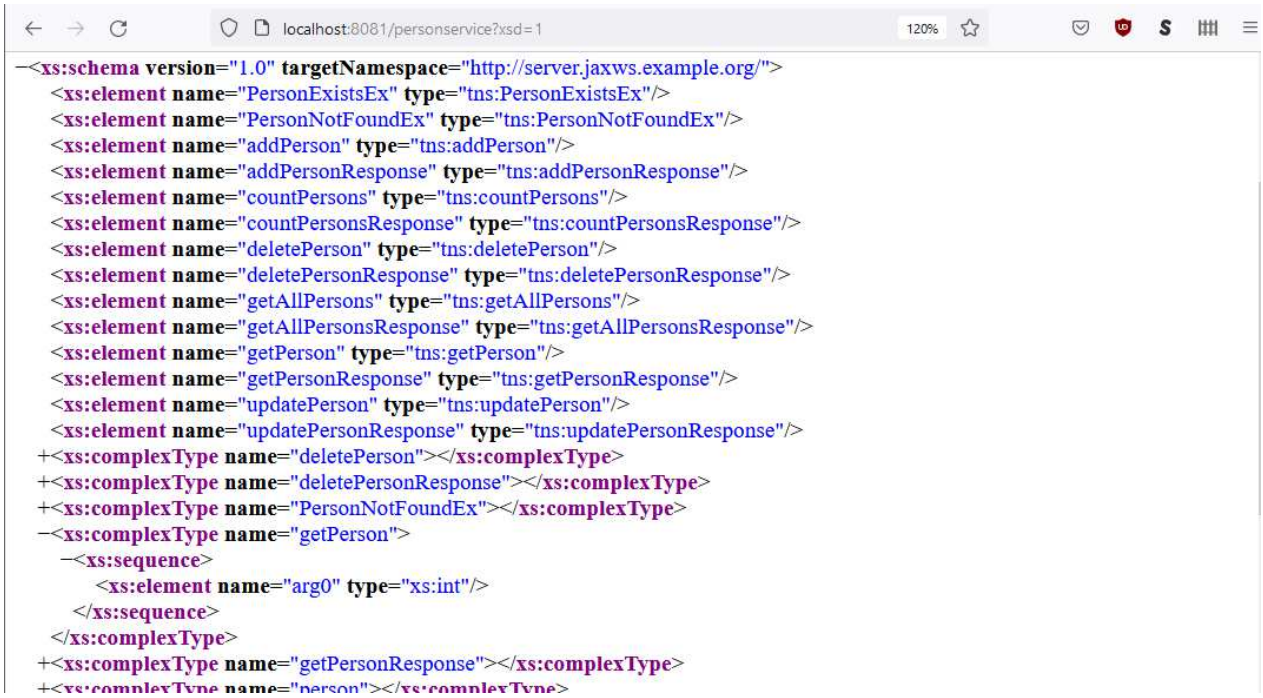


- Open the link of the WSDL data (here: **http://localhost:8081/personservice?wsdl**) and verify the content.



- Next open the link of the schema contained in the |WSDL data (here: **http://localhost:8081/personservice?xsd=1**).

  *Note: below in the figures most nodes are not unfolded –only one type (**getPerson** – composed of one integer(input parameter for **getPerson** message/method)) is unfolded.*



- Run the Postman app (or SOAP UI) and compose the **POST message** with the following settings

***Attention**: for an unknown reason parameters received from Postman massages are null. The operations with contain input parameters will be tested from java client. The instruction will be corrected when the problem will be solved.*

  - The **address** of the service (as implemented for the endpoint).
  - The **Content-type** header equal **text/xml**.
  - The **SOAPAction** header – should be equal to SOAP Action attribute of the message in WSDL but in this case it not works so can be set to anything.

  - In the body enter the SOAP xml message to call **getAllPersons** method (the Header node can be omitted):

```
<?xml version="1.0" encoding="utf-8"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <getAllPersons xmlns="http://server.jaxws.example.org/">
    </getAllPersons>
  </s:Body>
</s:Envelope>
```

As the result you should receive the list of person data.

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getAllPersonsResponse xmlns:ns2=...>
      <return>
        <age>9</age>
        <firstName>Marius</firstName>
        <id>1</id>
      </return>
      <return>
        <age>10</age>
        <firstName>Andrew</firstName>
        <id>2</id>
      </return>
      [...]
    </ns2:getAllPersonsResponse>
  </S:Body>
</S:Envelope>
```

3.2 Web Service JAX-WS client

The first step of creating Web Service JAX-WS client is the first step creating Web Service using **Top-Down method**.

3.2.1 WSDL file

The WSDL file will be used to generate interfaces and classes used to build Web Service and Web Service client.

- Add to the project the file (here named **personservice.wsdl**) in which description of service will be included – e.g. in newly created *resource* folder in *src/main/java* folder.
- Run the service created in section 3.1. Open the browser and open the link of the WSDL data.
  - Copy the whole content (you can omit comments) to the personservice.wsdl file.
  - Open the XSD schema data – the link is in the beginning of WSDL data in the **<import …/>** element inside **<types>** element.
  - Copy the whole content of schema data (you can omit comments) in place of **<types>** section in personservice.wsdl file.
  - Remove all **wsam:Action** attributes (only these attributes and its values) inside **<portType>** section.
  - Correct the **<definitions>** tag to make all data consistent:
    - the Schema, SOAP, and general WSDL namespaces.

    Here:

```
<definitions targetNamespace="http://server.jaxws.example.org/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://server.jaxws.example.org/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             name="PersonService">
```

3.2.2 POM file

In the POM file the plugin that uses **wsimport** tool and generates necessary interfaces and classes must be added.

- Open POM file and enter behind **<dependiences>** the **<build>** section and inside **com.sun.xml.ws/jaxws-maven-plugin** plugin with **wsimport** *goal* and proper *configuration*.

```
<build>
    <plugins>
      <plugin>
        <groupId>com.sun.xml.ws</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <version>3.0.2</version>
        <executions>
          <execution>
            <goals>
              <goal>wsimport</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <wsdlFiles>
            <wsdlFile>
              ${project.basedir}/src/main/resources/personservice.wsdl
            </wsdlFile>
          </wsdlFiles>
          <packageName>org.example.jaxws.server_topdown</packageName>
          <sourceDestDir>${project.basedir}/src/main/java/</sourceDestDir>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

The classes will be generated in **org.example.jaxws.server_topdown** package on the basis of the WSDL file specified in **<wsdlFile>** element., in **<sourceDestDir>** folder.

- Compile the project and verify project structure and generated classes.

3.2.3 The Web Service Client

The client uses generated ***ServiceName*_Service** class (where ServiceName is the name of defined service interface) to get proxy client used to call service operations.

- Create the new class (here in the separate client package) for the client. In the code:

- define URL of the service,
- create PersonService_Service object,
- get client proxy from PersonService_Service object,
- use client proxy to call Web Service operations.

```
package org.example.jaxws.client;
import [...]
public class ESClient {
  public static void main(String[] args) throws MalformedURLException,
                                                 PersonNotFoundEx_Exception
  {
    int num =-1;
    URL addr = new URL("http://localhost:8081/personservice?wsdl");
    PersonService_Service pService = new PersonService_Service();
    PersonService pServiceProxy = pService.getPersonServiceImplPort();
    num = pServiceProxy.countPersons();
    System.out.println("Num of Persons = "+num);
    Person person = pServiceProxy.getPerson(2);
    System.out.println("Person "+person.getFirstName()+",
                       id = "+person.getId());
    ...
  }
}
```

3.3 W3C Web service – Top-Down method

3.3.1 WSDL file

The first step of creating Web Service using **Top-Down method** is to create WSDL file. It is used to generate necessary interfaces and classes as described in previous section (client section).

Here the WSDL file already created for the client will be used.

3.3.2 POM file

The plugin necessary for building service interfaces and classes was already described in previous section (client section).

3.3.3 The service implementation

The generated classes defines only service interfaces, basic data structures, messages, etc. The service logic (implementation) must be defined similarly as for Bottom-Up method. With proper organization of packages in this exercise it could be the same classes. However, it was used separate **org.example.jaxws.server_topdown** package to present build the service in different way.

Here will be created in this package: Repository interface and class and service implementation class (here named **PersonService2Impl**).

- Copy **PersonRepository** class to the package. You may rename it to **PersonRepository2** (to distinct them).
  - Correct the code to use classes only from **org.example.jaxws.server_topdown** package.
  - Use **PersonNotFoundEx_Exception** and **PersonExistsEx_Exception** instead previous ones (because we use only classes generated from the WSDL file).
- Copy **PersonRepositoryImpl** class to the package. You may rename it to **PersonRepository2Impl** (to distinct them).
  - Do the same corrections as a moment before (to use classes from …**…server_topdown** package only).
  - You may remove annotations as they are not used now.

3.3.4 Service host

The service host (here named **ServiceHost2**) is almost the same as previously. The only difference is to use classes from **org.example.jaxws.server_topdown** package – precisely **PersonService2Impl**.

3.3.5 Running and testing the service

- Compile and run the application.
  Be aware that if you have not defined other service address (use the same address as first service) you can't run both at the same time.
- Run the client to test operation.

  As the client uses the same classes it may be used to test the service.
  Be aware of service address if you used the other one than previously.

4 Exercise – Part II

The detailed and final requirements for Part II are set by the teacher.

A. Develop or prepare to develop a program according to the teacher's instructions.