# Use case

In this section, we will be putting all that we have learned so fare. We will design and implement a **WSDL** web service first. A Simple use case that we are going to use is:

* Creating a **CustomerOrdersService** which allows the consumers to :
  + To create orders for a given customer.
  + Read the orders for a given customer.

## Creation Steps:

1. Coming up with a **WSDL** file where will be defining what types of messages we are going to exchange as well as what type of operations that is to create and get operations that allowing the customers to consume our web services.
2. To generate the stubs from this WSDL.
3. Implement our web services endpoint using the generated stubs.
4. Configure the web services point in the **cxf.services.xml, which** is a **spring** configuration file for **Apache CXF**.

# Steps to create WSDL First Web Service

1. **Create the Project:** there are two **additional steps** in case of **WSDL** first web services.
   1. **Creating the WSDL file**
   2. **Generating the Stubs from the WSDL file.**
2. Create the endpoint
3. Create the configuration class
4. Run the application

# Create the WSDL First project

1. We create a new simple **spring boot** project.
2. Adding Apache CXF dependency **(“org.apache.cxf”).**
3. Setting the application properties as the same old project (previous section).

# WSDL Creation

The first step in creating a contract first web service is to come up with **WSDL** file. In this lecture, we will see how easily come up with a WSDL file for our customer order service. That will allow the web service consumers to create customer order as well as retrieve, read the customer order back.

## WSDL Description

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **What’s WSDL** | It is an XML document. Therefore, we will be applying all the knowledge from the XML section. | | | | | | | | | |
| **element** | **definition** | | | | | | | | | |
| **<wsdl:defintions>**  **<wsdl:defintions>** | This is the root element. That is where we will define all the namespace that we want to use in this document. | | | | | | | | | |
| **namespace** | | | **The use** | | | | | | |
| **<xmlns:xsd>** | | | **XML** schema namespace. | | | | | | |
| **<xmlns:wsdl>** | | | **WSDL** namespace. | | | | | | |
| **<xmlns:soap>** | | | **SOAP** namespace. | | | | | | |
| **<targetNamespace>** | | | Our own namespace, which should apply for all the requests and responses. | | | | | | |
| **<name>** | | | To give a user a unique name (“**CustomerOrdersService**”) | | | | | | |
| **<wsdl:types>**  **<wsdl:types>** | The type section. This where we will spend most of the time in designing a **WSDL**. We define all the request and response types using an XML schema.  In our case we are going to allow the web service consumers to :   * Create orders. * Retrieve orders. | | | | | | | | |
| **Complex Types** | | | | | | | | |
| **Complex type** | | | **description** | | | | | |
| **Order** | | | Comprised of id and an unlimited number of products. | | | | | |
| **Product** | | | Comprised of id, description and quantity. | | | | | |
| **getOrdersRequest** | | | | By sending a customer id to web service provider, we should get back a list of orders. | | | | |
| **getOrdersResponse** | | | We use unbounded orders to get orders response (unlimited number of orders). | | | | | |
| **createOrdersRequest** | | | We send a customer id along with the order (we are allowing there one single order to send). | | | | | |
| **createOrdersResponse** | | | Simply is a Boolean value (whether the order is created or not) | | | | | |
| **Elements** | | | | | | | | |
| **element** | | | **Description** | | | | | |
| **getOrdersRequest** | | | Is of type getOrdersRequest (defined previously). | | | | | |
| **getOrdersResponse** | | | Is of type getOrdersResponse (defined previously). | | | | | |
| **createOrdersRequest** | | | Is of type getOrdersRequest (defined previously). | | | | | |
| **createOrdersResponse** | | | Is of type getOrdersResponse (defined previously). | | | | | |
| **<wsdl:message>** | We use the types defined in the previous section (**Elements**) to create the messages. These messages are analogous to the input parameters **Java** methods and output return types. | | | | | | | | |
| **<wsdl:part>** | Each parameters associates with a concrete type defined in the type’s container element. | | | | | | | |
| **getOrdersRequest** | | | | | Will be used to come up the operations. | | |
| **getOrdersResponse** | | | | | Will be used to come up the operations. | | |
| **createOrdersRequest** | | | | | Will be used to come up the operations. | | |
| **createOrdersResponse** | | | | | | Will be used to come up the operations. | |
| **<wsdl:portType>** | Combines multiple message elements to form a complete one way or round-trip operation | | | | | | | | |
| **<wsdl:operation>** | Defines each operation that the portType exposes. | | | | | | | |
| **operation** | | | **input** | | | | **output** |
| **getOrders** | | | getOrdersRequest | | | | getOrdresResponse |
| **createOrders** | | | | createOrdersRequest | | | createOrdersResponse |
| **<wsdl:binding>** | The physical portion, which tells how to consume this particular web service. We define a **SOAP** binding by giving it a unique name and we link it with the section above by using the **<portType>,** which we have defined here. The recommended SOAP binding is **document** **literal** for all operations because it allows the SOAP engines to validate the entire SOAP message going out and coming in. | | | | | | | | |
|  | **<wsdl:service>** | Tells the consumers how to consume this web service. We give it a unique name inside which we have a port, which contains actual **URL** of the web service. The **URL** will be replaced dynamically by Apache **CXF** | | | | | | | | |

# Generate the Stubs (Java Classes)

We are going to use Apache CXF WSDL, which is a Java tool that it provides. (**CXF codegen plugin**). These plugin needs some parameters:

|  |  |  |  |
| --- | --- | --- | --- |
| **<sourceRout>** | This is where all the generated stubs will be copied. | | |
| **<wsdlOptions>** | **<wsdlOption>** | **<wsdl>** | This where we can put the WSDL fil directly under our project (base directory). |
| **<wsdlLocation>** | It will try to find the **WSDL** file under a folder called **WSDL**. |

# Create the CustomerOrders Service

We will create the web service endpoint itself. If you look at the Stub’s (Java classes), each one of them is from a WSDL file. Therefore, each element that exists inside the WSDL file will be converted to a Java class.

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | | **Description** | |
| **All of these classes was generated from the WSDL file.** | | | |
| **CreateOrdersRequest** | | Marked automatically with the JAXB annotations. It has a customer id and an order. | |
| **CustomerOrdersPortType** | | It is an existed element inside the WSDL file. It is an Interface marked with a ton of JAX-WS annotations and our endpoint should implement. | |
| **package-info** | **ObjectFactory** | | CXF uses to create the generated classes. |
| **Order** | POJO class marked with JAXB annotations. | | |

The key here is to know that every element in **WSDL** file is converted into Java object (Java class, interface …).

# Implement the init method

In this lecture, we will create an in-memory database that will create a hash map inside this web services endpoint, which will act as our database and carry the customer and order information. We will initialize this hash map to handle a one order for each customer. Inside the init method, we will create a list of orders that has a products inside (we can create multiple orders and products as well). This method will be invoked inside the **CustomerOrdersWebServiceImpl** constructor.

# Implement the getOrders method

In this lecture, we will implement the getOrders method. This method is going to take a customer id from the request and send the orders list in the response. (We give the parameters a meaningful name).

# Implement the createOrders method

In this lecture, we will implement the createOrders method. We are going to retrieve the customer id and the order from the request and we will add it to the existing orders of that customer. In general,

* We take the customer id and the order.
* We add the order to customer map.
* We send back a Boolean result
  + “**True**” if the creation process has succeeded
  + “**False**” if the creation process has failed.

# Publish the endpoint (Already done in PART 7)

In this step, we are going to configure our web service endpoint (publish our web service endpoint) by creating a Java Spring configuration file.

# Enable Logging Feature (Already done in PART 7)

In this step, we are going to enable the logging feature for our project to make the possibility for **CXF** to log all the incoming and outgoing **SOAP** messages (Already done).

# Run the application

We will run **WSDL FIRST** project and see the list of services from our web browser.

# Test using SoapUI

In this step, we are going to test our **WSDL** **FIRST** web service by using **soapUI**.

|  |  |
| --- | --- |
| **Operation** | **Steps** |
| **getOrders** | * Sending the customer id (SOAP message as XML format) to web service. * The web services uses getOrders method to fetch the aimed data. * Getting back the list of orders as a response (SOAP message as XML format). |
| **createOrders** | * Sending an order element with the customer id to the web service. * CreateOrders method handles the inputs by inserting the order to existing list. * Getting back a result that tells whether the creation process was successful or not (**true** or **false**). |

# WSDL First Web Service Assignment

In this lecture, we are going to implement the delete orders method. The process will go under the following steps:

1. Starting with **WSDL** file, we will define two new complex types (**DeleteOrdersRequest** and **DeleteOrdersResponse**).
2. In the messages section, we use the types defined previously.
3. Inside the portType element, we have to define the operations (**deleteOrders**) which will use **deleteOrdersRequest** as input and **DeleteOrdersResponse** as output.
4. When we save the **WSDL**, the plugin will generate automatically the new Java class under the dedicated folder.
5. Finally, we have to add the newly generated methods inside **CustomerOrdersPortType** interface and implementing the new business logic for deleting orders.

# Section Summary

In this section, we have created an order processing **WSDL** **FIRST** web service from scratch.

**Table des matières**

[1. Use case 1](#_Toc94648797)

[1.1. Creation Steps: 1](#_Toc94648798)

[2. Steps to create WSDL First Web Service 1](#_Toc94648799)

[3. Create the WSDL First project 1](#_Toc94648800)

[4. WSDL Creation 1](#_Toc94648801)

[4.1. WSDL Description 1](#_Toc94648802)

[5. Generate the Stubs (Java Classes) 3](#_Toc94648803)

[6. Create the CustomerOrders Service 3](#_Toc94648804)

[7. Implement the init method 3](#_Toc94648805)

[8. Implement the getOrders method 3](#_Toc94648806)

[9. Implement the createOrders method 3](#_Toc94648807)

[10. Publish the endpoint (Already done in PART 7) 4](#_Toc94648808)

[11. Enable Logging Feature (Already done in PART 7) 4](#_Toc94648809)

[12. Run the application 4](#_Toc94648810)

[13. Test using SoapUI 4](#_Toc94648811)

[14. WSDL First Web Service Assignment 4](#_Toc94648812)

[15. Section Summary 4](#_Toc94648813)