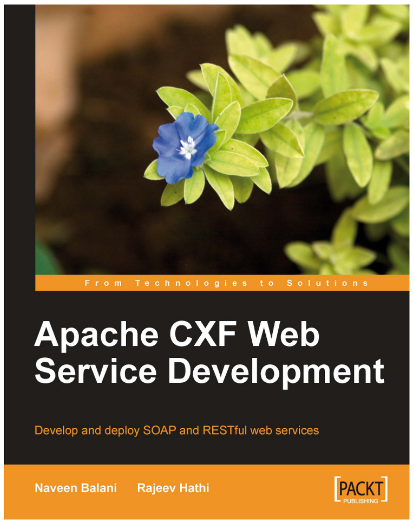
Apache CXF Web Service Development



Naveen Balani

Rajeev Hathi

Chapter No. 2
"Developing a Web Service with CXF"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.2 "Developing a Web Service with CXF"

A synopsis of the book's content

Information on where to buy this book

About the Authors

Naveen Balani works as a Software Architect with IBM India Software Labs (ISL). He leads the design and development activities for Web Sphere Business Services Fabric product out of ISL Mumbai. He has over nine years of industrial experience and has architected and implemented large scale enterprise solutions.

Naveen Balani likes to research upcoming technologies and is a Master Author with IBM developer Works having written over 60 plus publications, on topics such as Web services, ESB, JMS, SOA, architectures, open source frameworks, semantic Web, J2ME, pervasive computing, Spring, Ajax, and various IBM products. He started working with web services way back in 2001 and proposed the first MVC web services-based pattern (`http://www.ibm.com/developerworks/library/ws-mvc/`)in 2002.

Naveen Balani's articles on Spring Series (`http://www.ibm.com/developerworks/web/library/wa-spring1/`) were rated as the top articles in the last 10 years for developer Works web architecture zone. He has co-authored books on Spring framework (`http://www.wrox.com/WileyCDA/WroxTitle/Beginning-Spring-Framework-2.productCd-047010161X.html`) and Multiple IBM Redbooks on Web Sphere Business Services Fabric and BPM 6.2 Product deployments. You can reach him on his website—`http://soaweb.co.in`

I would like to thank my wonderful wife, Sonia, for her love and patience and her endless support in spending many hours sitting beside me, reviewing my work and providing valuable inputs.

I would also like to thank my parents for their support and encouragement in all my endeavors.

And last but not least, to my good friend and co-author Rajeev Hathi.

Rajeev Hathi is a J2EE Consultant and Developer living in Mumbai, India. He grew up in a joint Hindu family and pursued his primary education in the field of Economics and Commerce. His hobbies are watching sports and listening to rock music. His favorite bands are Pink Floyd and Dire Straits.

Rajeev has written several articles for IBM developer Works portal. His major contributions are in the fields of Java, web service, and DB2. He developed an interest in computers after pursuing a diploma in Advanced Systems Management at NIIT (National Institute of Information Technology).

Rajeev has been working on J2EE-based projects for more than ten years now. He has worked with several companies offering software services and conducted various knowledge sessions on Java and J2EE. He has attained several Java-based certifications such as SCJP, SCWCD, SCBCD, and SCEA. He, along with the co-author Naveen Balani, has initiated a portal `http://soaweb.co.in` which aims to provide online consulting on the subject of web services.

A book is often the product of many hands. To start with I'd like to thank Usha Iyer, an Acquisition Editor with Packt Publishing, for having enough faith in my writing skills and abilities. My special thanks to the Packt Publishing team in making enormous efforts to make this book a reality. A good book cannot be made better without a constructive review and feedback and the reviewers equally contributed to the whole writing process.

I owe thanks to my wonderful and lovely friend, Sunita, who instilled in me enough confidence and zest to make my writing look effortless. I owe thanks and gratitude to my family members who have supported and encouraged my writing efforts day and night. And last but not least, without my co-author and amazing friend Naveen Balani, this project would not have been achievable.

Finally, I would like to dedicate this book to my late parents and late sister without their blessings, this project would have just remained a mere thought.

Apache CXF Web Service Development:

Apache CXF is an open source services framework that makes web service development easy, simplified, and standard based. CXF provides many features such as frontend programming, support for different transports and data bindings, support for different protocols, and other advanced concepts like Features and Invokers. It also provides a programming model to build and deploy RESTful services.

The focus of the book is to provide readers with comprehensive details on how to use the CXFframework for web services development. The book begins by giving us an overview of CXF features and architecture. Each feature is explained in a separate chapter, each of which covers well defined practical illustrations using real world examples. This helps developers to easily understand the CXF API. Each chapter provides hands on examples and provides step-by-step instructions to develop, deploy, and execute the code.

What This Book Covers

The book is about the CXF service development framework. The book covers two of the most widely used approaches, for web services development, SOAP and REST. Each chapter in the book provides hands on examples, where we look in detail at how to use the various CXF features in detail to develop web services in a step-by-step fashion

Chapter 1: *Getting Familiar with CXF* revisits web service concepts and provides an introduction to CXF framework and its usage, and prepares the CXF environment for the following chapters. By the end of this chapter the reader will be able to understand the core concepts of CXF.

Chapter 2: *Developing a Web Service with CXF* focuses on getting the reader quickly started with the CXF framework by developing a simple web service and running it under the Tomcat container.

By the end of this chapter the reader will be able to develop a simple web service using CXF.

Chapter 3: *Working with CXF Frontends* illustrates the use of different frontends, like JAX-WS and CXF simple fronted API, and shows how to apply code-first and contract-first development approaches for developing web services. We will look at how to create dynamic web service clients, the use of web service context, and how to work directly with XML messages using CXF Provide and Dispatch implementation.

By the end of this chapter the reader will be able to apply different frontends to develop a web service.

Chapter 4: *Learning about Service Transports* explains basic transport protocols for a service and shows you how to configure HTTP, HTTP(s), JMS, and Local protocol for web services communication. You will get introduced to the concept of HTTP conduit, which enables the client program to apply policies or properties to HTTP and HTTPs protocols, and how to generate a crypto key and a key store for HTTPs based service communication. You will learn how to use JMS protocol for web services communication and how to facilitate web services message exchange using CXF Local service transport.

By the end of this chapter the reader will be able develop services with different transports

Chapter 5: *Implementing Advanced Features* will explain advanced concepts using CXF Features, Interceptors, and Invokers, and how to integrate these concepts in existing applications.

By the end of this chapter the reader will be able develop services with features like Interceptors and Invokers

Chapter 6: *Developing RESTful Services with CXF* explains the concept of REST technology and JAX-RS specifications, how CXF realizes the JAX-RS specification, and demonstrates additional features for developing enterprise RESTful services. We will look at how to design, develop, and unit test the RESTful Service by taking a real world example using CXF JAX-RS implementation.

By the end of this chapter the reader will be able to design, develop, and unit test the RESTful service

Chapter 7: *Deploying RESTful Services with CXF* will explain how to deploy REST services in a container like Tomcat using Spring configuration, and how to test out the various operations exposed by the RESTFul application using CXF RESTful client API using a web service development tool. We will look at how to enable exception handling, JSON message support, and logging support for RESTful applications using CXF framework.

By the end of this chapter the reader would be able utilize various CXF features for developing RESTful services and how to leverage Spring configuration for deploying RESTful service in the tomcat container.

Chapter 8: *Working with CXF Tools* will explain some of the commonly used CXF tools that assist us in web services development. We will look at how to invoke a real world .NET service over the internet using a Java client and JavaScript, create web service implementation from WSDL files, generate WSDL files from web service implementation, and validate the WSDL file for compliance.

By the end of this chapter the reader will be able to use different CXF tools to develop a service.

Appendix A deals with how to set up the CXF environment, provides details on how the source code for each chapter is organized, and shows how to run the source code examples using the ANT tool and Maven Tool.

Appendix B provides an explanation of the basics of the Spring framework and IoC concepts, along with an end-to-end example which utilizes Spring IoC concepts.

By the end of this Appendix chapter the reader will have a good understanding of Spring capabilities used in the context of CXF web services development in this book.

2
Developing a Web Service with CXF

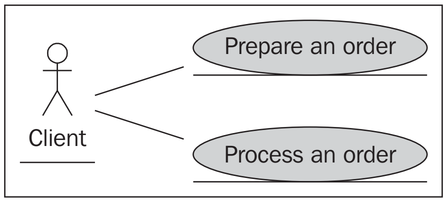
The first chapter provided an introduction to web services and CXF framework. We looked at the features supported by the CXF framework and how to set up the CXF environment. This chapter will focus on programming web service with CXF. CXF provides a robust programming model that offers simple and convenient APIs for web service development. The chapter will focus on illustrating a simple web service development using CXF and Spring-based configurations. The chapter will also talk about the architecture of CXF.

Before we examine CXF-based web service development, we will review the example application that will be illustrated throughout the book. The example application will be called **Order Processing Application**. The book will demonstrate the same application to communicate different concepts and features of CXF so that the reader can have a better understanding of CXF as a whole. This chapter will focus on the following topics:

- Overview of a sample Order Processing Application
- CXF-based web service development with Spring
- Insight into CXF architecture

The Order Processing Application

The objective of the Order Processing Application is to process a customer order. The order process functionality will generate the customer order, thereby making the order valid and approved. A typical scenario will be a customer making an order request to buy a particular item. The purchase department will receive the order request from the customer and prepare a formal purchase order. The purchase order will hold the details of the customer, the name of the item to be purchased, the quantity, and the price. Once the order is prepared, it will be sent to the Order Processing department for the necessary approval. If the order is valid and approved, then the department will generate the unique order ID and send it back to the Purchase department. The Purchase department will communicate the order ID back to the customer.



For simplicity, we will look at the following use cases:

- Prepare an order
- Process the order

The client application will prepare an order and send it to the server application through a business method call. The server application will contain a web service that will process the order and generate a unique order ID. The generation of the unique order ID will signify order approval.

In real world applications a unique order ID is always accompanied by the date the order was approved. However, in this example we chose to keep it simple by only generating order ID.

Developing a service

Let's look specifically at how to create an Order Processing Web Service and then register it as a Spring bean using a JAX-WS frontend.

In Chapter 3 you will learn about the JAX-WS frontend. The chapter will also cover a brief discussion on JAX-WS. The Sun-based JAX-WS specification can be found at the following URL:

`http://jcp.org/aboutJava/communityprocess/final/jsr224/index.html`

JAX-WS frontend offers two ways of developing a web service—Code-first and Contract-first. We will use the *Code-first* approach, that is, we will first create a Java class and convert this into a web service component. The first set of tasks will be to create server-side components.

In web service terminology, Code-first is termed as the Bottoms Up approach, and Contract-first is referred to as the Top Down approach.

To achieve this, we typically perform the following steps:

- Create a **Service Endpoint Interface** (**SEI**) and define a business method to be used with the web service.
- Create the implementation class and annotate it as a web service.
- Create `beans.xml` and define the service class as a Spring bean using a JAX-WS frontend.

Creating a Service Endpoint Interface (SEI)

Let's first create the SEI for our Order Processing Application. We will name our SEI `OrderProcess`. The following code illustrates the `OrderProcess` SEI:

```
package demo.order;

import javax.jws.WebService;

@WebService
public interface OrderProcess {
  @WebMethod
  String processOrder(Order order);
}
```

As you can see from the preceding code, we created a Service Endpoint Interface named `OrderProcess`. The SEI is just like any other Java interface. It defines an abstract business method `processOrder`. The method takes an Order bean as a parameter and returns an order ID String value. The goal of the `processOrder` method is to process the order placed by the customer and return the unique order ID.

One significant thing to observe is the `@WebService` annotation. The annotation is placed right above the interface definition. It signifies that this interface is not an ordinary interface but a web service interface. This interface is known as **Service Endpoint Interface** and will have a business method exposed as a service method to be invoked by the client.

The `@WebService` annotation is part of the JAX-WS annotation library. JAX-WS provides a library of annotations to turn Plain Old Java classes into web services and specifies detailed mapping from a service defined in WSDL to the Java classes that will implement that service. The `javax.jws.WebService` annotation also comes with attributes that completely define a web service. For the moment we will ignore these attributes and proceed with our development.

The `javax.jws.@WebMethod` annotation is optional and is used for customizing the web service operation. The `@WebMethod` annotation provides the operation name and the action elements which are used to customize the `name` attribute of the operation and the SOAP action element in the WSDL document.

The following code shows the `Order` class:

```
package demo.order;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "Order")
public class Order {

   private String customerID;
   private String itemID;
   private int qty;
   private double price;

   // Contructor
   public Order() {
   }

   public String getCustomerID() {
      return customerID;
   }

   public void setCustomerID(String customerID) {
      this.customerID = customerID;
   }
```

```
        public String getItemID() {
            return itemID;
        }
        public void setItemID(String itemID) {
            this.itemID = itemID;
        }
        public int getQty() {
            return qty;
        }
        public void setQty(int qty) {
            this.qty = qty;
        }
        public double getPrice() {
            return price;
        }
        public void setPrice(double price) {
            this.price = price;
        }
    }
```

As you can see, we have added an `@XmlRootElement` annotation to the `Order` class. The `@XmlRootElement` is part of the **Java Architecture for XML Binding** (**JAXB**) annotation library. JAXB provides data binding capabilities by providing a convenient way to map XML schema to a representation in Java code. The JAXB shields the conversion of XML schema messages in SOAP messages to Java code without having the developers know about XML and SOAP parsing. CXF uses JAXB as the default data binding component.

The `@XmlRootElement` annotations associated with `Order` class map the `Order` class to the XML root element. The attributes contained within the `Order` object by default are mapped to `@XmlElement`. The `@XmlElement` annotations are used to define elements within the XML. The `@XmlRootElement` and `@XmlElement` annotations allow you to customize the namespace and name of the XML element. If no customizations are provided, then the JAXB runtime by default would use the same name of attribute for the XML element. CXF handles this mapping of Java objects to XML.

Developing a service implementation class

We will now develop the implementation class that will realize our `OrderProcess` SEI. We will name this implementation class `OrderProcessImpl`. The following code illustrates the service implementation class `OrderProcessImpl`:

```
@WebService
public class OrderProcessImpl implements OrderProcess {

    public String processOrder(Order order) {
      String orderID = validate(order);
        return orderID;
    }

   /**
    * Validates the order and returns the order ID
   **/
    private String validate(Order order) {
      String custID = order.getCustomerID();
      String itemID = order.getItemID();
      int qty = order.getQty();
      double price = order.getPrice();

      if (custID != null && itemID != null && !custID.equals("")
                         && !itemID.equals("") && qty > 0
                         && price > 0.0) {
         return "ORD1234";
      }
      return null;
   }
}
```

As we can see from the preceding code, our implementation class `OrderProcessImpl` is pretty straightforward. It also has `@WebService` annotation defined above the class declaration. The class `OrderProcessImpl` implements `OrderProcess` SEI. The class implements the `processOrder` method. The `processOrder` method checks for the validity of the order by invoking the `validate` method. The `validate` method checks whether the `Order` bean has all the relevant properties valid and not null.

It is recommended that developers explicitly implement OrderProcess SEI, though it may not be necessary. This can minimize coding errors by ensuring that the methods are implemented as defined.

Next we will look at how to publish the OrderProcess JAX-WS web service using Sping configuration.

Spring-based server bean

What makes CXF the obvious choice as a web service framework is its use of Spring-based configuration files to publish web service endpoints. It is the use of such configuration files that makes the development of web service convenient and easy with CXF.

[Please refer to the *Getting Started with Spring framework* appendix chapter to understand the concept of Inversion of Control, **AOP** (**Aspect oriented program**), and features provided by the Spring framework using a sample use case.]

Spring provides a lightweight container which works on the concept of **Inversion of Control** (**IoC**) or **Dependency Injection** (**DI**) architecture; it does so through the implementation of a configuration file that defines Java beans and its dependencies. By using Spring you can abstract and wire all the class dependencies in a single configuration file. The configuration file is often referred to as an Application Context or Bean Context file.

We will create a server side Spring-based configuration file and name it as `beans.xml`. The following code illustrates the `beans.xml` configuration file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
   xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

   <import resource="classpath:META-INF/cxf/cxf.xml" />
   <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
   <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

   <jaxws:endpoint
     id="orderProcess"
     implementor="demo.order.OrderProcessImpl"
     address="/OrderProcess" />

</beans>
```

Let's examine the previous code and understand what it really means. It first defines the necessary namespaces. It then defines a series of `<import>` statements. It imports `cxf.xml`, `cxf-extension-soap.xml`, and `cxf-servlet.xml`. These files are Spring-based configuration files that define core components of CXF. They are used to kick start CXF runtime and load the necessary infrastructure objects such as WSDL manager, conduit manager, destination factory manager, and so on

The `<jaxws:endpoint>` element in the `beans.xml` file specifies the `OrderProcess` web service as a JAX-WS endpoint. The element is defined with the following three attributes:

- `id`—specifies a unique identifier for a bean. In this case, `jaxws:endpoint` is a bean, and the `id` name is `orderProcess`.
- `implementor`—specifies the actual web service implementation class. In this case, our implementor class is `OrderProcessImpl`.
- `address`—specifies the URL address where the endpoint is to be published. The URL address must to be relative to the web context. For our example, the endpoint will be published using the relative path `/OrderProcess`.

The `<jaxws:endpoint>` element signifies that the CXF internally uses JAX-WS frontend to publish the web service. This element definition provides a short and convenient way to publish a web service. A developer need not have to write any Java class to publish a web service.

Developing a client

In the previous section we discussed and illustrated how to develop and publish a web service. We now have the server-side code that publishes our `OrderProcess` web service. The next set of tasks will be to create the client-side code that will consume or invoke our `OrderProcess` web service. To achieve this, we will perform the following steps:

- Develop the `client-beans.xml` to define the client factory class as a Spring bean using JAX-WS frontend
- Develop a client Java application to invoke the web service

Developing a Spring-based client bean

We will create a client-side Spring-based configuration file and name it as `client-beans.xml`. The following code illustrates the `client-beans.xml` configuration file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxws="http://cxf.apache.org/jaxws"
   xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
<jaxws:client id="orderClient" serviceClass=
                 "demo.order.OrderProcess" address=
                 "http://localhost:8080/orderapp/OrderProcess" />
</beans>
```

The `<jaxws:client>` element in the `client-beans.xml` file specifies the client bean using JAX-WS frontend. The element is defined with the following three attributes:

- `id`—specifies a unique identifier for a bean. In this case, `jaxws:client` is a bean and the `id` name is `orderClient`. The bean will represent an SEI.
- `serviceClass`—specifies the web service SEI. In this case our SEI class is `OrderProcess`
- `address`—specifies the URL address where the endpoint is published. In this case the endpoint is published at the URL address: `http://localhost:8080/orderapp/OrderProcess`

`<jaxws:client>` signifies the client bean that represents an `OrderProcess` SEI. The client application will make use of this SEI to invoke the web service. Again, CXF internally uses JAX-WS frontend to define this client-side component.

Developing web service client code

We will now create a standalone Java class to invoke our `OrderProcess` web service. The following code illustrates the client invocation of a web service method:

```
public final class Client {
    public Client() {
    }
    public static void main(String args[]) throws Exception {
        // START SNIPPET: client
        ClassPathXmlApplicationContext context
```

```
                = new ClassPathXmlApplicationContext(new String[]
                      {"demo/order/client/client-beans.xml"});

            OrderProcess client = (OrderProcess) context.
                                   getBean("orderClient");
          // Populate the Order bean
          Order order = new Order();
          order.setCustomerID("C001");
          order.setItemID("I001");
          order.setQty(100);
          order.setPrice(200.00);
            String orderID = client.processOrder(order);
            String message = (orderID == null) ?
                              "Order not approved" : "Order approved;
                               order ID is " + orderID;
            System.out.println(message);
            System.exit(0);
```

As you can see from the above code, we have the `main` method that first loads the `client-beans.xml` configuration file. It uses the Spring application context component `ClassPathXmlApplicationContext` to load the configuration file. The context component's `getBean` method is passed the bean ID `orderClient`. This method will return the `OrderProcess` SEI component. Using the SEI, we then invoke the web service method `processOrder`. One thing to observe here is that the client always uses the interface to invoke a web service method. The `processOrder` method takes the `Order` bean as a parameter. The following code depicts the `Order` bean:

```
public class Order {

   private String customerID;
   private String itemID;
   private int qty;
   private double price;

   // Contructor
   public Order() {
   }

   // Getter and setter methods for the above declared properties
}
```

The above `Order` bean is populated with the valid values and passed to the `processOrder` method. The method will then process the order and return the unique order ID.

We have now finished developing server and client side components. To summarize, we created the `OrderProcess` service endpoint interface and the implementation class. We then created server and client-side Spring-based configuration files and finally we created the client application. The relevant components are developed and we are all set to run or execute our code. But before we do that, you will have to create one final component that will integrate Spring and CXF.

We need to wire Spring and CXF through `web.xml`. The following code illustrates the `web.xml` file:

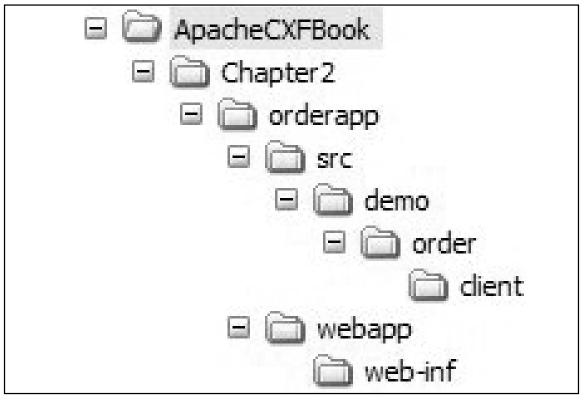
```
<web-app>
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>WEB-INF/beans.xml</param-value>
   </context-param>

   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>

   <servlet>
      <servlet-name>CXFServlet</servlet-name>
      <display-name>CXF Servlet</display-name>
      <servlet-class>
         org.apache.cxf.transport.servlet.CXFServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>CXFServlet</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Let's go through the above piece of code. The `web.xml`, as we know, is the web application configuration file that defines a servlet and its properties. The file defines `CXFServlet`, which acts as a front runner component that initiates the CXF environment. It defines the `listener` class `ContextLoaderListener`, which is responsible for loading the server-side configuration file `beans.xml`. So upon the web server startup, the order process web service endpoint will be registered and published.

Running the program

The source code and build file for the chapter is available in the `Chapter2/orderapp` folder of the downloaded source code.

Before running the program, we will organize the code so far developed in the appropriate folder structure. You can create the folder structure, as shown in the following screenshot, and put the components in the respective sub folders



The developed code will go into the following:

- The Java code will go into the respective package folders
- The `beans.xml` and `web.xml` will go into the `webapp\WEB-INF` folder
- The `client-beans.xml` file will go into the `demo\order\client` folder

Once the code is organized, we will go about building and deploying it in the Tomcat server. It will typically involve three steps:

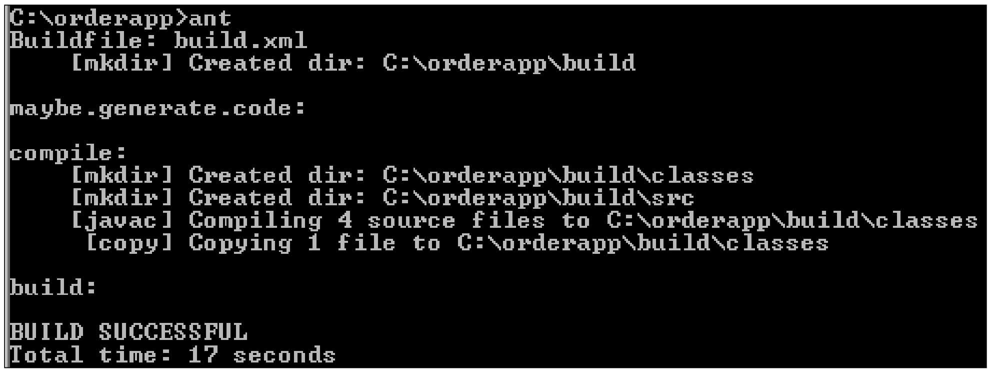
- Building the code
- Deploying the code
- Executing the code

Building the code

Building the code means compiling the source Java code. We will use the ANT tool to do this. The ANT file is provided in `Chapter2\orderapp` folder. The following code illustrates the sample `build.xml` build script:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="CXF Chapter2 example" default="build" basedir=".">
   <import file="common_build.xml"/>
    <target name="client" description=
                 "run demo client" depends="build">
        <property name="param" value=""/>
        <cxfrun classname="demo.order.client.Client" />
    </target>
    <target name="server" description=
                 "run demo server" depends="build">
        <cxfrun classname="demo.spring.servlet.Server"/>
    </target>
    <property name="cxf.war.file.name" value="orderapp"/>
      <target name="war" depends="build">
     <cxfwar filename="${cxf.war.file.name}.war" webxml=
                      "webapp/WEB-INF/web.xml" />
    </target>
</project>
```

Alongside `build.xml`, you will also find `common_build.xml` in the same folder. The `common_buid.xml` refers to `CATALINA_HOME` environment variable to find location of tomcat installation. Please make sure that you have set up the environment variables as mentioned in Appendix A. Open the command prompt window, go to `C:\orderapp` folder and run the **ant** command. It will build the code and put the class files under the newly created `build` folder. The following figure shows the output generated upon running the `ant` command.



```
C:\orderapp>ant
Buildfile: build.xml
    [mkdir] Created dir: C:\orderapp\build

maybe.generate.code:

compile:
    [mkdir] Created dir: C:\orderapp\build\classes
    [mkdir] Created dir: C:\orderapp\build\src
    [javac] Compiling 4 source files to C:\orderapp\build\classes
     [copy] Copying 1 file to C:\orderapp\build\classes

build:

BUILD SUCCESSFUL
Total time: 17 seconds
```

Deploying the code

Having built the code, we will deploy it. Deployment effectively means building and moving the code archive to the server deploy path. We will be using the Tomcat web container to deploy and run the application. To deploy our built code, navigate to `project root` folder, and enter the following command:

```
ant deploy
```

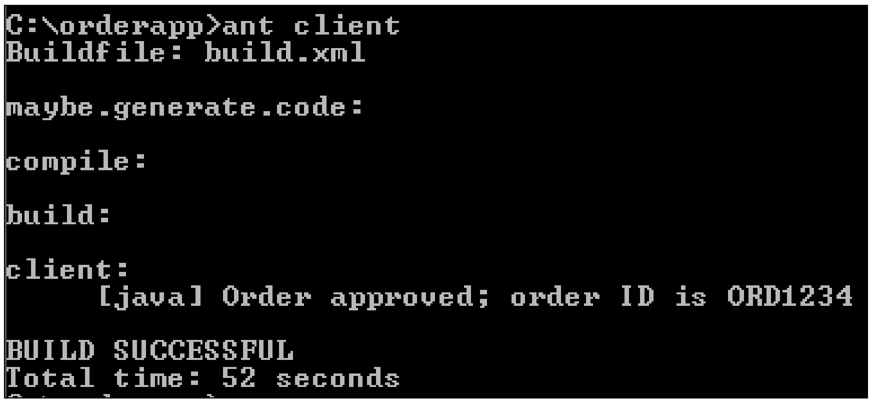
This will build the WAR file and put it under the Tomcat server `webapp` path. For example, if you have installed the Tomcat under the root folder, then the WAR will be deployed to `/Tomcat/webapp` folder.

Executing the code

Following code deployment, we are all set to run the Order Process Application. You will execute the Java client program `Client.java` to invoke the Order Process web service. The program will invoke the `processOrder` method that will generate the order ID if the specified order is approved. Before running the client program, we need to start the Tomcat web server. There are several ways of starting the Tomcat server depending on the Tomcat version that is installed. Once the server is started, you need to run the client program by giving the following command at the command prompt window:

```
ant client
```

As you can see above, we are using Ant to run the client program. Upon executing this command, it will generate the following output:



```
C:\orderapp>ant client
Buildfile: build.xml

maybe.generate.code:

compile:

build:

client:
     [java] Order approved; order ID is ORD1234

BUILD SUCCESSFUL
Total time: 52 seconds
```

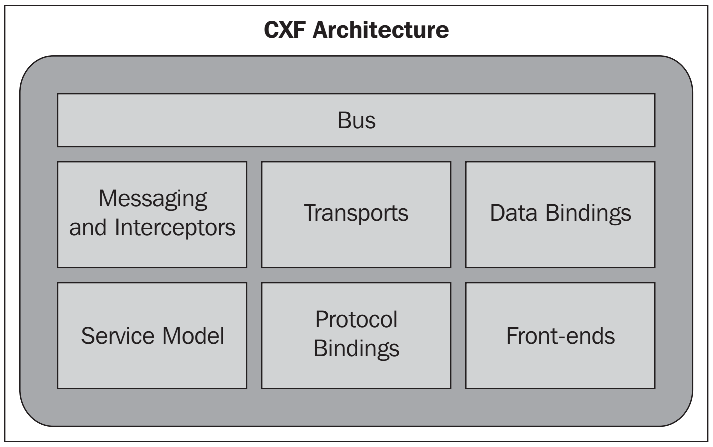
Thus we have successfully executed the order processing web service.

CXF architecture

The architecture of CXF is built upon the following components:

- Bus
- Frontend
- Messaging and Interceptors
- Service Model
- Data bindings
- Protocol bindings
- Transport

The following figure shows the overall architecture:



Bus

Bus is the backbone of the CXF architecture. The CXF bus is comprised of a Spring-based configuration file, namely, `cxf.xml` which is loaded upon servlet initialization through `SpringBusFactory`. It defines a common context for all the endpoints. It wires all the runtime infrastructure components and provides a common application context. The `SpringBusFactory` scans and loads the relevant configuration files in the `META-INF/cxf` directory placed in the classpath and accordingly builds the application context. It builds the application context from the following files:

- `META-INF/cxf/cxf.xml`
- `META-INF/cxf/cxf-extension.xml`
- `META-INF/cxf/cxf-property-editors.xml`

The XML file is part of the installation bundle's core CXF library JAR. Now, we know that CXF internally uses Spring for its configuration. The following XML fragment shows the bus definition in the `cxf.xml` file.

```
<bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl" />
```

The core bus component is `CXFBusImpl`. The class acts more as an interceptor provider for incoming and outgoing requests to a web service endpoint. These interceptors, once defined, are available to all the endpoints in that context. The `cxf.xml` file also defines other infrastructure components such as `BindingFactoryManager`, `ConduitFactoryManager`, and so on. These components are made available as bus extensions. One can access these infrastructure objects using the `getExtension` method. These infrastructure components are registered so as to get and update various service endpoint level parameters such as service binding, transport protocol, conduits, and so on.

CXF bus architecture can be overridden, but one must apply caution when overriding the default bus behavior. Since the bus is the core component that loads the CXF runtime, many shared objects are also loaded as part of this runtime. You want to make sure that these objects are loaded when overriding the existing bus implementation.

You can extend the default bus to include your own custom components or service objects such as factory managers. You can also add interceptors to the bus bean. These interceptors defined at the bus level are available to all the endpoints. The following code shows how to create a custom bus:

```
SpringBeanFactory.createBus("mycxf.xml")
```

`SpringBeanFactory` class is used to create a bus. You can complement or overwrite the bean definitions that the original `cxf.xml` file would use. For the CXF to load the `mycxf.xml` file, it has to be in the classpath or you can use a factory method to load the file. The following code illustrates the use of interceptors at the bus level:

```
<bean id="cxf" class="org.apache.cxf.bus.spring.SpringBusImpl">
    <property name="outInterceptors">
           <list>
              <ref bean="myLoggingInterceptor"/>
           </list>
        </property>
</bean>
<bean id="myLogHandler" class="org.mycompany.com.cxf.logging.
                               LoggingInterceptor">
    ...
</bean>
```

The preceding bus definition adds the logging interceptor that will perform logging for all outgoing messages.

Frontend

CXF provides the concept of frontend modeling, which lets you create web services using different frontend APIs. The APIs let you create a web service using simple factory beans and JAX-WS implementation. It also lets you create dynamic web service clients. The primary frontend supported by CXF is JAX-WS. We will look at how to use the Frontend programming model in the next chapter.

JAX-WS

JAX-WS is a specification that establishes the semantics to develop, publish, and consume web services. JAX-WS simplifies web service development. It defines Java-based APIs that ease the development and deployment of web services. The specification supports WS-Basic Profile 1.1 that addresses web service interoperability. It effectively means a web service can be invoked or consumed by a client written in any language. JAX-WS also defines standards such as **JAXB** and **SAAJ**. CXF provides support for complete JAX-WS stack.

JAXB provides data binding capabilities by providing a convenient way to map XML schema to a representation in Java code. The JAXB shields the conversion of XML schema messages in SOAP messages to Java code without the developers seeing XML and SOAP parsing. JAXB specification defines the binding between Java and XML Schema. SAAJ provides a standard way of dealing with XML attachments contained in a SOAP message.

JAX-WS also speeds up web service development by providing a library of annotations to turn Plain Old Java classes into web services and specifies a detailed mapping from a service defined in WSDL to the Java classes that will implement that service. Any complex types defined in WSDL are mapped into Java classes following the mapping defined by the JAXB specification.

As discussed earlier, two approaches for web service development exist: Code-First and Contract-First. With JAX-WS, you can perform web service development using one of the said approaches, depending on the nature of the application.

With the Code-first approach, you start by developing a Java class and interface and annotating the same as a web service. The approach is particularly useful where Java implementations are already available and you need to expose implementations as services.

You typically create a **Service Endpoint Interface** (**SEI**) that defines the service methods and the implementation class that implements the SEI methods. The consumer of a web service uses SEI to invoke the service functions. The SEI directly corresponds to a `wsdl:portType` element. The methods defined by SEI correspond to the `wsdl:operation` element.

```
@WebService
public interface OrderProcess {
    String processOrder(Order order);
}
```

JAX-WS makes use of annotations to convert an SEI or a Java class to a web service. In the above example, the `@WebService` annotation defined above the interface declaration signifies an interface as a web service interface or Service Endpoint Interface.

In the Contract-first approach, you start with the existing WSDL contract, and generate Java class to implement the service. The advantage is that you are sure about what to expose as a service since you define the appropriate WSDL Contract-first. Again the contract definitions can be made consistent with respect to data types so that it can be easily converted in Java objects without any portability issue. In Chapter 3 we will look at how to develop web services using both these approaches.

WSDL contains different elements that can be directly mapped to a Java class that implements the service. For example, the `wsdl:portType` element is directly mapped to SEI, type elements are mapped to Java class types through the use of **Java Architecture of XML Binding** (**JAXB**), and the `wsdl:service` element is mapped to a Java class that is used by a consumer to access the web service.

The `WSDL2Java` tool can be used to generate a web service from WSDL. It has various options to generate SEI and the implementation web service class. As a developer, you need to provide the method implementation for the generated class. If the WSDL includes custom XML Schema types, then the same is converted into its equivalent Java class.

[In Chapter 8 you will learn about CXF tools. The chapter will also cover a brief discussion on the *wsdl2java* tool.]

Simple frontend

Apart from JAX-WS frontend, CXF also supports what is known as 'simple frontend'. The simple frontend provides simple components or Java classes that use reflection to build and publish web services. It is simple because we do not use any annotation to create web services. In JAX-WS, we have to annotate a Java class to denote it as a web service and use tools to convert between a Java object and WSDL. The simple frontend uses factory components to create a service and the client. It does so by using Java reflection API. In Chapter 3 we will look at how to develop simple frontend web services

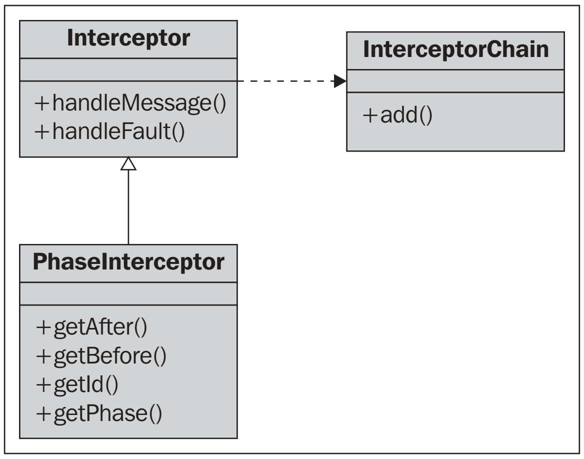
The following code shows a web service created using simple frontend:

```
// Build and publish the service
OrderProcessImpl orderProcessImpl = new OrderProcessImpl();
ServerFactoryBean svrFactory = new ServerFactoryBean();
svrFactory.setServiceClass(OrderProcess.class);
svrFactory.setAddress("http://localhost:8080/OrderProcess");
svrFactory.setServiceBean(orderProcessImpl);
svrFactory.create();
```

Messaging and Interceptors

One of the important elements of CXF architecture is the Interceptor components. Interceptors are components that intercept the messages exchanged or passed between web service clients and server components. In CXF, this is implemented through the concept of Interceptor chains. The concept of Interceptor chaining is the core functionality of CXF runtime.

The interceptors act on the messages which are sent and received from the web service and are processed in chains. Each interceptor in a chain is configurable, and the user has the ability to control its execution.



The core of the framework is the Interceptor interface. It defines two abstract methods—`handleMessage` and `handleFault`. Each of the methods takes the object of type `Message` as a parameter. A developer implements the `handleMessage` to process or act upon the message. The `handleFault` method is implemented to handle the error condition. Interceptors are usually processed in chains with every interceptor in the chain performing some processing on the message in sequence, and the chain moves forward. Whenever an error condition arises, a `handleFault` method is invoked on each interceptor, and the chain unwinds or moves backwards.

Interceptors are often organized or grouped into phases. Interceptors providing common functionality can be grouped into one phase. Each phase performs specific message processing. Each phase is then added to the interceptor chain. The chain, therefore, is a list of ordered interceptor phases. The chain can be created for both inbound and outbound messages. A typical web service endpoint will have three interceptor chains:

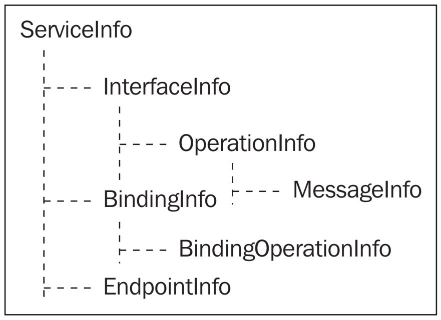
- Inbound messages chain
- Outbound messages chain
- Error messages chain

There are built-in interceptors such as logging, security, and so on, and the developers can also choose to create custom interceptors.

In Chapter 5 we will learn about working with CXF advanced features. The chapter will mainly focus on Interceptors.

Service model

The **Service model**, in a true sense, models your service. It is a framework of components that represents a service in a WSDL-like model. It provides functionality to create various WSDL elements such as operations, bindings, endpoints, schema, and so on. The following figure shows the various components that form the Service model:



The components of the Service model can be used to create a service. As you can see from the above figure, the service model's primary component is **ServiceInfo** which aggregates other related components that make up the complete service model. **ServiceInfo** is comprised of the following components that more or less represent WSDL elements:

- **InterfaceInfo**
- **OperationInfo**
- **MessageInfo**
- **BindingInfo**
- **EndpointInfo**

A web service is usually created using one of the frontends offered by CXF. It can be either constructed from a Java class or from a WSDL.

CXF frontends internally use the service model to create web services. For example, by using a simple frontend, we can create, publish, and consume web services through factory components such as `ServerFactoryBean` and `ClientProxyFactoryBean`. These factory classes internally use the service model of CXF.

Data binding

Data binding is the key for any web service development. Data binding means mapping between Java objects and XML elements. As we know, with web service, messages are exchanged as XML artifacts. So there has to be some way to convert these XML into Java objects and vice versa for the application to process as service and client. Data binding components perform this mapping for you. CXF supports two types of data binding components—**JAXB** and **Aegis**. CXF uses JAXB as the default data binding component. As a developer, you have the choice of specifying the binding discipline through a configuration file or API. If no binding is specified, then JAXB is taken as a default binding discipline. The latest version of CXF uses JAXB 2.1. JAXB uses annotations to define the mapping between Java objects and XML. The following code illustrates the use of JAXB annotations:

```
@XmlRootElement(name="processOrder", namespace=" http://localhost/
                      orderprocess")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name="processOrder", namespace=
              " http://localhost/orderprocess")
public class OrderProcess {
    @XmlElement(name="arg0", namespace="")
    private order.Order arg0;
   //Gettter and Setter
….
}
```

As shown in the previous code, the `@Xml` specific annotations represents the JAXB metadata that is used by JAXB to map Java classes to XML schema constructs. For example, the `@XmlType` annotation specifies that the `OrderProcess` class will be mapped to complex XSD element type '`processOrder`' that contains an element '`arg0`' of type '`Order`' bean.

CXF also supports the Aegis data binding component to map between Java objects and XML. Aegis allows developers to gain control of data binding through its flexible mapping system. You do not have to rely on annotations to devise the mapping. Your Java code is clean and simple POJO.

Aegis also supports some annotations that can be used to devise binding. Some of the annotations that can be used with Aegis are:

- `XmlAttribute`
- `XmlElement`
- `XmlParamType`
- `XmlReturnType`
- `XmlType`

In Aegis, you define the data mapping in a file called `<MyJavaObject>.aegis.xml`, where `MyJavaObject` is the object that you are trying to map with XML. Aegis reads this XML to perform the necessary binding. Aegis also uses reflection to derive the mapping between Java object and XML. The following code fragment shows the sample Aegis mapping file:

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings>
    <mapping name="HelloWorld">
        <method name="sayHi">
            <parameter index="0" mappedName=
                             "greeting" nillable='false' />
        </method>
    </mapping>
</mappings>
```

The above XML fragment states that a string parameter of a method named `sayHi` of the bean `HelloWorld` should be mapped to a name as `greeting`.

You can configure your web service to use Aegis data binding as follows:

```
<jaxws:endpoint id="orderProcess" implementor="demo.order.
OrderProcessImpl" address="/OrderProcess" >
   <jaxws:dataBinding>
    <bean class="org.apache.cxf.aegis.databinding.AegisDatabinding" />
   </jaxws:dataBinding>
</jaxws:endpoint>
```

Protocol binding

Bindings bind the web service's messages with the protocol-specific format. The messages, in web service terminology, are nothing but an operation with input and output parameters. The message defined in the web service component is called a logical message. The logical message used by a service component is mapped or bound to a physical data format used by endpoints in the physical world. It lays down rules as to how the logical messages will be mapped to an actual payload sent over the wire or network.

Bindings are directly related to port types in a WSDL artifact. Port types define operations and input and output parameters which are abstract in nature. They define the logical message, whereas binding translates this logical message into actual payload data defined by the underlying protocol. The following WSDL portion shows the sample binding details:

```
<wsdl:binding name="OrderProcessImplServiceSoapBinding"
              type="tns:OrderProcess">
   <soap:binding style="document" transport=
                    "http://schemas.xmlsoap.org/soap/http" />
   <wsdl:operation name="processOrder">
      <soap:operation soapAction="" style="document" />
      <wsdl:input name="processOrder">
         <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="processOrderResponse">
         <soap:body use="literal" />
      </wsdl:output>
     </wsdl:operation>
</wsdl:binding>
```

As you can see from the above sample binding fragment, it is defined using the `<binding>` element. This element has two attributes, namely, `name` and `type`. The `name` attribute identifies the binding, and the `type` attribute maps it with the port type. The `name` attribute of the binding element is used to associate with the endpoint. The child elements of the `<binding>` parent element define the actual mapping of the messages with the protocol format. In the previous case, the communication protocol used is SOAP 1.1.

CXF supports the following binding protocols:

- SOAP 1.1
- SOAP 1.2
- CORBA
- Pure XML

Transports

Transport defines the high-level routing protocol to transmit the messages over the wire. Transport protocols are associated with the endpoints. One endpoint can communicate with another using a specific transport protocol. Transport details are nothing but networking details. Service endpoints are a physical representation of a service interface. Endpoints are composed of binding and networking details. In a WSDL artifact, transport details are specified as part of the `<port>` element. The port element is a child of the service element. The WSDL portion following shows the sample transport details:

```
<wsdl:service name="OrderProcessImplService">
   <wsdl:port binding="tns:OrderProcessImplServiceSoapBinding"
              name="OrderProcessImplPort">
      <soap:address location="http://localhost:8080/orderapp/
                              OrderProcess" />
   </wsdl:port>
</wsdl:service>
```

As you see from the above XML fragment, transport details are specified as part of the service element. The service element has one child element as port element. The port element maps to binding as defined by the binding element and provides details of the transport. The previous example shows SOAP as binding protocol and HTTP as a transport protocol. In Chapter 4, the various transport protocols are explained in the context of web services development.

CXF supports the following transports for its endpoints:

- HTTP
- CORBA
- JMS
- Local

Summary

The chapter started by describing the Order Processing Application and we saw how to develop a web service with CXF and Spring-based configuration. CXF's seamless integration with Spring makes it extremely easy and convenient to build and publish a web service. We also saw how to build, deploy, and execute a web service using ANT and Tomcat. The chapter later described the CXF architecture, which is built upon the core components. These components lay down the foundation for building web services.