# Payment Gateway Legacy Application Use Case

In this lecture, we will be setting up a context that is required for the developing of java first web service. We will develop a payment gateway web service (a very simple class and we are going to expose it out as a web service so that consumers can consume it).

|  |  |  |
| --- | --- | --- |
| **Example** | Let’s say Amazon has asked us as a bank that they want to process their payments through the payment gateway of our bank. This whole application (as you can see) the layers of the Java EE application: |  |
| This entire application was completely written for Amazon. However, in the future, we have EBay, Flipkart, and several other web sites who want to do the same payment processing through our bank. Therefore, we have to rewrite the whole logic for those particular clients. So instead of doing that, we will expose out a payment gateway web service. Therefore, we will take the business layer, we annotate it and mark it as Java First web service, and then eBay and Flipkart will be able to call our web service to process the payment. So this a very good example when we need to use Java first web service when the code is already exists (a Legacy application is already there) and we want to expose out that legacy service as a web service to allow other applications to use it. | |

# Import the legacy project

The steps that are required to expose out a web service using Apache CXF are:

1. Creating the Project.
2. Create the web service endpoint.
3. Mark the beans or DTO’s (the Data Transfer Objects) with JAXB annotations.
4. Mark the web service endpoint classes with JAXWS annotations.
5. Create the configuration file to publish the web service endpoint.
6. Run the application.

# Mark the beans with JAXB Annotations

They key work is to mark PaymentProcessorRequest class with the JAXB annotations and then the fields on it so that the objects of this class can be serialized into xml and deserialized back.

1. We put the annotation **@XmlType** on the main class (PaymentProcessorRequest). Then the class will have its own complex type in the generated WSDL and eventually in the SOAP message request.
   1. The annotation **@XmlType** Optionally can have a name (by default the class name will be used as the XML complex type or the element).
2. We need to mark the fields with the **@XmlElement** annotation but we need to tell the JAXB processor whether we are marking it at the field level or the setter getter level.
   1. The annotation **@XmlAccessorType** tells the JAXB runtime that is the Apache CXF in our case at which level the JAXB Annotations are present in the class. Therefore, inside the brackets, we use XmlAccessType.FIELD.
   2. We are going to mark our fields with JAXB Annotations. So each field here will becomes an element in the XML. (We can optionally give it a name & we can specify whether the field is required or not by using, a flag (attribute) called required. By default, this flag is true).

# Mark the Endpoint with JAX-WS annotations

In this step, we are going to mark our java classes with JAX-WS annotations so they will become web services endpoints.

1. We mark simply the interface PaymentProcessor with **@WebService** annotation from JAX-WS API. This tells the Apache CXF or any SOAP run time that this particular class or interface should be exposed as web service. Optionally, we can give it a name; this name will be reflected in the WSDL file at runtime. By default, the name of the class or the interface is the same. In our case, we take the class name and past it into the attribute called name. That is the only annotation that is required to expose this as a web service.
2. The method does not need to be marked with any annotations but optionally you can use at **@WebParam** annotation on the request. We can control the name of the request in the WSDL file and in the SOAP message.
3. We mark the response using **@WebResult** annotation from JAX-WS and we give that a name.
4. Optionally, we can mark the web service method with **@WebMethod.**

# Publish the endpoint

In this step, we need to create a configuration class so that our web service will be published. We need also to add an application context for our application to do that.

1. We need to create a configuration class called WebServiceConfig:
   1. Inside this class, we have an endpoint method, we have an object called endpoint, and it should have as an Endpoint implementation. We need to expose here the payment processor implementation.
   2. The URL should be /PaymentProcessor. We can use any URL or URI.
   3. We need to create a dedicated package for this configuration class (as a good practice).
2. Now, we have to go the application properties inside the resources directory, there we can configure root application context.

# Run the application

What exactly happens when this application runs is **CXF**

1. Through **spring** will scan our classes.
2. Figure out which classes are marked with **JAX-WS** annotations
3. Based on that annotation, it will build a **WSDL** file on the fly.
4. Each element from the **WSDL** file is generated from our **Java** classes.
5. The services page is being exposed out by **CXF** automatically.

# Test Using SoapUI

When the request is sent:

1. Apache CXF receives the request
2. It serializes or deserializes that request into the payment processor request.
3. It pass the generated object to the web service implementation.
4. It will takes the object serialize it into XML or SOAP message.
5. Send it back to the client.

# Create a Code First Web Service

# SOAPUI Assignment (Testing steps)

# Section Summary

In this section, we have use of the Java First web services or code first web services to expose out the legacy payment gateway application as a web service. We have done that in three steps:

1. Coming up with the endpoints and beans.
2. Annotating the endpoints and beans with **JAXWS** and **JAXB** annotations.
3. Publishing our endpoint with the CXF **configuration** inside the (**application.properties**).
4. Running our application on the server and **CXF** generates the **WSDL** file for us on the fly.
5. Testing our web service using the **WSDL** file with soapUI tool.