# Introduction

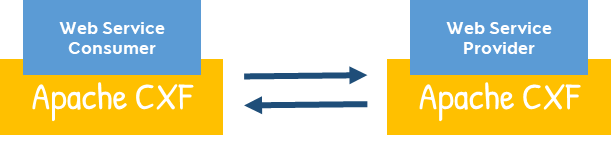
* A web services stack or engines provide us with various tools that we use to develop our web services.
* They also serialize and deserialize the upcoming requests into language objects and back on the wire to the **XML** or **JSON** formatted messages.
* They dispatch the incoming web services requests from the clients to the appropriate web services classes and methods that we code (to handle the requests).

There are many of them but we are interesting in **Apache** **CXF** and we will learn how to create maven project using **Apache** **CXF** from scratch.

# Apache CXF Overview

## Web Service Engine

The various tools provided by web services stacks or engines allow us:



* To build and run web services.
* As developers to focus on the business logic and the application itself.

Using **Apache** **CXF**, we can develop both the web services **providers** and web services **consumers** for both **SOAP** as well as **restful** web services.

## What is CXF?

Apache CXF is one of the powerful and popular services engines in the java space because it implements both **JAXWS** and **JAXRS**. Using **Apache** **CXF**, we can develop both the web services providers and web services consumers for both **SOAP** and **Restful** web services.

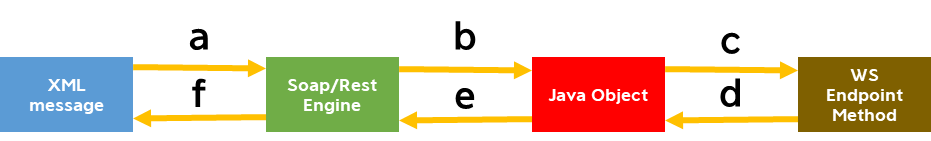
## Why CXF?

CXF comes with a **SOAP** and **Rest** engine, which at a minimum do two things:

1. It serializes the **XML** or **JSON** or any other formatted messages requests and responses into **Java** objects. It deserialize the Java objects to **XML** or **JSON**.
2. It dispatches the incoming request to the appropriate web service endpoint.

When an **XML** message comes in:

1. The engine takes that web service message request.
2. It converts the message to Java object.
3. Hands the Java object to an appropriate method on the web service endpoint.
4. It then takes the response from these web services classes method
5. It converts it back into **XML** or **JSON** or any other formatted format.
6. Finally, it send it back to the client.



## Web Service Standards

It implements almost all the Web services standards out there like:

* WS-Security
* WS-Transaction Management
* WS-Policy

We can simply configure these standards to our application using configuration files.

## Tools

We need tools to develop web services providers and consumers.

|  |  |
| --- | --- |
| **WSDL2JAVA** | Allow us to generate code from a **WSDL** file and implement our web services provider. It will also allow us to implement a web service client from these stubs that get generated using **WSDL2JAVA**. |
| **JAVA2WSDL** | Allow us to code first, development from **Java** to **WSDL**. |
| **Note** | |
| We can build our stubs automatically using mavin plugin. | |

## Spring configuration

It can be done with two ways:

1. By annotation
2. **CXF** uses **Spring** to configure all the web services endpoints and the other features.

## Extend and Customize

It also easy to extend and customize **CXF** using the interceptors or handlers **JAX**-**WS** handlers. If we want to add custom things, that **CXF** does not have.

## Documentation and Samples

**CXF** comes with a lot of documentation on their website and it uses a lot of samples examples. We can visit the offical web site for more information. (http://cxf.apache.org/)

# CXF and Spring Boot

Apache **CXF** makes it super easy to create web service applications by supporting the use of **spring boot.** Once we start the use of **spring** boot with **CXF** we have:

1. We simply add a **CXF Spring** dependency (a maven dependency) to the main pom.xml. Once we add this dependency and automatically all the other jar files that are required to build **RESTful** services or **SOAP** services will be transitively dragged in.
2. It also makes the publishing of the **RESTful** endpoint very easy. We simply add a property called

**cxf.jaxrs.component-scan = true**. Once we do that in the application.properties (which is the spring boot properties file), automatically all your web services endpoints will be discovered and published. Therefore, there is no need for **Java** configuration.

1. We can add the server context path. Using this, you can define a web application context for your **RESTful** or **SOAP** application.

# Spring Boot

# Create the SOAP project

In this and the next few lectures, we will create a simple Hello World SOAP web service in four simple steps:

1. Create the Project (by creating this **spring** **boot** project).
2. Create the web service endpoint class itself.
3. Create the configuration via **Spring** **Java** configuration class (the class will publish our endpoint).
4. We run the application and view the **WSDL** being generated.

# Create the endpoint

To create a web service endpoint:

1. Inside the main source folder, we need to create a Java class named **HelloWorldWebService**.
2. We add to the last class a **String** method with the name **sayHelloWorld,** which returns a string (“**Hello World from HelloWorldWebService**”).

**Note:** The class is just a simple Java class (POJO),

1. To make it a SOAP endpoint, we mark it using **@WebService** annotation from the **JAX-WS** **API**.
2. Now we have to annotate the method with **@WebMethod** always from the **JAX-WS** **API**.

**Note:** if you go to and look at the maven dependencies, you will see all the **CXF** dependencies.

# Create the configuration class

In this step, we are going to create a Java Spring configuration, which tells **Apache** **CXF** about our endpoint and at which **URL** this endpoint should be available. We need to suit the following steps:

1. We create a class named **WebServiceConfig** inside a **config** package under the **main** package.
2. We mark this class using **@Configuration** annotation from the spring framework packages. This tells **spring** that this particular **bean** or **Java** **class** is a **Java** **configuration** file.
3. Inside this class, we will define a method, which will return an endpoint. We need to import the Endpoint from the JAX packages (**CXF** provides an implementation for this interface).

# Run the application

We run the application as a **Spring** **boot** application.

1. We need to go to localhost: 8080.
2. By default, the CXF exposes all the services at the URL called /services.
3. If we access that, CXF generates a beautiful page on the fly that displays all the services available (In our case, it is the HelloWorldWebService and it has a method called sayHelloWorld).
4. To access the WSDL file of that web service, you can click on the link (we will able to download the entire **WSDL** file).
5. We can grab this **WSDL** link and go to any tool like **soapUI** and we will be able to run the test.
6. We did not define any web application context (you do not see your project name here). The **HelloWorldWebService** is not being used in the **URL**.
7. We can define the last step as a spring boot property.
8. We can also tell CXF that we do not need this service.

# Change the web application context

We have seen that **spring** **boot** by default will not use the application name as the **context**. By default, Apache CXF uses a relative URI called /services. Therefore, in the lecture we will see how to configure a web application context or name and how to get rid of this /services if want to.

Inside the file called **application.properties, (**you will find it under src/main/resources**)** which is the **spring boot** properties file:

|  |  |
| --- | --- |
| **Properties** | **Description** |
| **server.servlet.context-path** | To define a new path for your web application (in our case it is **hellowebservice**). |
| **cxf.path** | Tells the Apache CXF at runtime that to do not use **/services** by exposing all my endpoints at the root level itself. |
| **Note** | |
| We need to restart the application after each modification on the **application.properties** file. | |

# Test using SoapUI

One last step is to test our SOAP web service. To handle the test:

1. We need to grab the WSDL file by copying it from browser.
2. We run soapUI tool (**SOAP** **web services** testing tool).
3. We create a new **SOAP** project by pasting the **WSDL** **URL** inside the initial **WSDL** field.
4. Automatically the project name will be picked up (change it to **helloWorldWebService**).

|  |  |
| --- | --- |
| **Checkbox** | **Description** |
| **Create Requests** | Create sample request for all operations |
| **Create TestSuite** | Create a **TestSuite** for the imported **WSDL** |
| **Relative Paths** | Stores all file paths n project relatively to project file (requires save). |

1. The tool generates a sample request dynamically.
2. To send the request make sure that the application is running to see a response back from web service **(“Hello World from HelloWorldWebService class”).**

The general steps to handle the request are:

1. The message goes in by using the **HTTP** **POST** request.
2. On the server side, Apache CXF will grab this request and convert it into a Java Object
3. Apache CXF from the request message will know that it should invoke **sayHelloWorld** method on and an endpoint.
4. CXF takes the response, serializes it into a SOAP message, and sends it back to the client.

# Enable Logging Feature

**CXF** provides a logging feature to log the incoming SOAP requests. We do that by:

1. On the **HelloWorldWebService** class we add the annotation **@Features** provided from Apache CXF
2. We want to unable the logging feature by providing it as an attribute to the previous annotation.
3. We can find this logging feature on (“**org.apache.cxf.feature.LoggingFreature**”).
4. Once we do that, CXF will start logging the SOAP messages on the console.

# Using Spring Boot 2.X

**Apache CXF** web services project (apache.cxf.3.2.4) supports for the **Spring boot** 2.x is included. So:

* We need to use latest version of **spring boot 2.x**.
* We need to use an **Apache CXF 3.5.4** in order to support the latest version of **Spring boot.**

Once we have those dependencies, we have to clean up the repository folder by using maven clean and we need to regenerate the dependencies by using maven install.

# Change the Context Property

There are some changing properties inside application.properties file.

|  |  |
| --- | --- |
| **Old properties (deprecated)** | **New property** |
| server.context-path | server.servlet.context-path=/hellowebservice |

# Section Summary

In this section, we have learned that the Apache CXF is one of the most common web services frameworks or stacks in the Java World. It implements both **JAX-WS** and **JAX-RS** standards for **Oracle**. It provides tools like WSDL2JAVA and JAVA2WSDL to implement top down or contract first and code first web services. It comes with SOAP engine that can serialize and deserialize with a XML data. It also dispatches the incoming web services requests to the appropriate methods on Java endpoint classes. CXF implements also the various Web services standards like security, addressing and attachments, which make our job as developers easier to implement the non-functional requirements for our application. It has a huge user based documentation and samples.

1. Apache CXF

[1. Introduction 1](#_Toc94618023)

[2. Apache CXF Overview 1](#_Toc94618024)

[2.1. Web Service Engine 1](#_Toc94618025)

[2.2. What is CXF? 1](#_Toc94618026)

[2.3. Why CXF? 1](#_Toc94618027)

[2.4. Web Service Standards 1](#_Toc94618028)

[2.5. Tools 1](#_Toc94618029)

[2.6. Spring configuration 2](#_Toc94618030)

[2.7. Extend and Customize 2](#_Toc94618031)

[2.8. Documentation and Samples 2](#_Toc94618032)

[3. CXF and Spring Boot 2](#_Toc94618033)

[4. Spring Boot 2](#_Toc94618034)

[5. Create the SOAP project 2](#_Toc94618035)

[6. Create the endpoint 2](#_Toc94618036)

[7. Create the configuration class 2](#_Toc94618037)

[8. Run the application 3](#_Toc94618038)

[9. Change the web application context 3](#_Toc94618039)

[10. Test using SoapUI 3](#_Toc94618040)

[11. Enable Logging Feature 4](#_Toc94618041)

[12. Using Spring Boot 2.X 4](#_Toc94618042)

[13. Change the Context Property 4](#_Toc94618043)

[14. Section Summary 4](#_Toc94618044)