# What are SOAP Web Services?

“English to this world are what web services are to the computer world. SOAP[[1]](#endnote-1) is a common communication medium”. They enable loosely coupled & easy communication between applications running on different operating systems, which were built in different programming languages. It an XML[[2]](#endnote-2)-based communication protocol to allow application exchanging information over HTTP[[3]](#endnote-3).

|  |
| --- |
| Figure 1 : Simple communication between web application & browser |
| Figure 2: Communication between web services |

# What are advantages & disadvantages of SOAP Web Services?

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| * Platform independent: * HTTP (Transport Independent). * XML (Data Independent). * Application Tailoring/Customization. * Legacy Applications are great! * New Revenue/Profit Channels :   + Example: Siebel On demand Services.   Firewalls like web services. | Ambiguous Web Services Standards.  Performance Impact due to Serialization & Deserialization. |

# When to use SOAP Web Services?

We should use SOAP Web Services when a Formal Contract is required to describe the interface that the web service provides (as a WSLD[[4]](#endnote-4) file). We should use it also when our application address many non-Functional requirements such as Security, Transaction management. We use it also in application that provides a Reliable Asynchronous Processing (Messaging between a producer & consumer).

# What is SOAP?

The Simple Object Access Protocol (SOAP). It is a specification from W3C[[5]](#endnote-5).

A specification usually is a set of rules. In case of SOAP, these rules com in form of an XML document. W3C defines all elements that we can use to send our web services requests and receive the responses back.

# SOAP Structure

|  |  |
| --- | --- |
| Envelope | This mandatory element is the root of the SOAP message, identifying the transmitted XML as being a SOAP packet. An envelope contains a body section and an optional header section. |
| Header | This optional element provides an extension mechanism indicating processing information for the message. For example, if the operation using the message requires security credentials, those credentials should be part of the envelope header. |
| Body | This element contains the message payload, the raw data being transmitted between the sending and receiving applications. The body itself may consist of multiple child elements, with an XML schema typically defining the structure of this data. |
| Fault | An optional Fault element that provides information about errors that occur while processing the message (Exception Response). |
|  | |

# What is WSDL?

WSDL, or Web Service Description Language, is an XML based definition language. It is used for describing the functionality of SOAP based web service.

WSDL files are central to testing SOAP-based services. SoapUI uses WSDL files to generate test request, assertions and mock services. WSDL files define various aspects of SOAP messages:

* Whether any element or attribute is allowed to appear multiple times.
* The required or optional elements and attributes.
* A specific order of elements, if it is required.

You may consider a WSDL files as a contract between the provider and the consumer of the service.

# What are WSDL Document Elements?

## The Abstract part

|  |  |  |
| --- | --- | --- |
| **The Abstract Part** | | |
| Tells what this web service provides. | | |
| **definitions** | Contains the definition of one or more services. | |
| **types** | In this section, we define all the data types, which we need to exchange information for this particular web service. |
| **message** | An abstract definition of the data being communicated. |
| **Operation** | An abstract description of the action supported by the service. |
| **portType** | It is simply a container of all the operations that your web service is providing. |
| **port** | Specifies a single endpoint as an address for the binding, thus defining a single communication endpoint. |
| **service** | Specifies the port address (es) of the binding. The service is a collection of network endpoints or ports. |

## The Physical part

|  |  |  |
| --- | --- | --- |
| **The Physical Part** | | |
| Tell how to consume this web service from the consumer. | | |
| **binding** | It tells how the consumer can consume this web service and how the provider is going to send the responses back. | |
| **soap** **binding** | By default is document literal wrapped (this is the recommended binding because XML engines like **apache CXF[[6]](#endnote-6)**) can validate that entire soap message. Therefore, for each operation, we can define binding. |
| **service** | It tells how to access this web service that actually **URL** inside a sub element called port. Therefore, we define a **PORT** for each web service and a **URL** to access that web service. You can have multiple web services inside a single user using multiple port elements. The **URL** location here will be replaced by the server IP address and the web applications name relatively far during runtime by **Apache CXF** or any other web services engine. | |

# What are the WSDL Binding Styles?

We can specify a soap binding style by using the style attribute on the soap-binding element in a WSDL file, for example “document/literal” is one the styles.

There are three important things that will be impacted depending on the style will choose. The first one is how the soap payload the message inside. The second one is whether the entire soap body can be validated against the schema that will use in the WSDL is determined by the time we use. The last one is whether the operation name (the name of the method) (web service method that should be invoked) is a part of the soap Body. This one is also determined by that document literal style.

|  |
| --- |
| RPC[[7]](#endnote-7)/encoded |
|  |
| Strengths |
|  |
| Weaknesses |
|  |
| RPC/literal |
|  |
| Strengths |
|  |
| Weaknesses |
|  |
| Document/encoded |
|  |
| Strengths |
|  |
| Weaknesses |
|  |
| Document/encoded |
|  |
| Strengths |
|  |
| Weaknesses |
|  |
| Document/literal wrapped |
|  |
| Strengths |
|  |
| Weaknesses |
|  |

# What are SOAP Web Services Design approaches?

## Top Down

|  |
| --- |
| **Top Down (WSDL First) (Contract First)** |
| * We first create the WSDL file (contract between web services provider & consumer). * We generate the Java stubs using tools like WSDL2JAVA. Therefore, we generate java classes from the WSDL file which we have created in the first step. * We implement the web services endpoint. It’s the last and the most important step, by implementing web services classes |
| **Advantages** |
| * Contract with the consumer Signed off. * Better interoperability. For example, if you end up using a data structure, which is that in a language you are using like Java and if very old legacy application is trying to consume this application and that language does not have the support for the data structure your web service, will become useless. * Faster integration. |

## Bottom Up

|  |
| --- |
| **Code First or Bottom Up** |
| * We write Java Code and annotate. * We generate the WSDL from the code using JAVA2WSDL. |
| **Advantages** |
| * Legacy Application Integration. |

## Which design to choose?

We should use Contract first as much as possible especially if we are developing web services from Scratch. For the Other Approach (Code First), we use it only when we work with legacy applications and we need to expose out some functionality in the legacy application as a web service so that it can be consumed by other applications, which want to use it.

# What is JAX-WS?

|  |  |  |  |
| --- | --- | --- | --- |
| **JAX-WS[[8]](#endnote-8)** | | | |
| JAX-WS stands for Java API or XML based Web services. Like any other Java enterprise standard comprises of specification and an API. | | | |
| **Specification** | | A set of rules or guidelines from Oracle. When they write the JAX-WS standard, the guidelines in the specification help web services engines like Apache CXF and glassfish to implement the JAX-WS standards. | |
| **API** | | It is for the developers and it comprises a set of Java Annotations. We as developers once we learn these annotations and we simply mark our Java classes and methods with these annotations and we are done we can implement both web services providers and consumers using these annotations. All these annotations are runtime annotations so CXF can read these annotations on our classes and methods | |
| **Annotations** | | | |
| It is for the developers and it comprises a set of Java Annotations. We as developers once we learn these annotations and we simply mark our Java classes and methods with these annotations and we are done we can implement both web services providers and consumers using these annotations. All these annotations are runtime annotations so CXF can read these annotations on our classes and methods | | | |
| **CORE** | All the annotations are contained inside java.jws package. | | |
| @WebService | | | When we mark our Java class or interface with this particular annotation, the web services engines know that this class or interface is an endpoint web services. |
| @WebMethod | | | We use this against all of our web services methods each method in our web services endpoint. |
| @WebFault | | | Come up with our custom exceptions, which in turn will be converted, into soap faults. |
| @SOAPBinding | | | It allows developers to specify a particular type of binding. Binding controls how SOAP message is generated when it goes on the wire apache CXF. By default is document\literal (recommended approach). We can specify a different type of binding by simply parametrizing this annotation. |
| @RequestWrapper | | | It allows us to wrap or map the incoming SOAP message to our Java objects in a customer manner and the response wrapper does the other way around. |
| @ResponseWrapper | | | It allows us to customize the way our response Java object is converted into a SOAP message. We very rarely do this because the JAX WS standard already does a great job in converting the SOAP messages into Java objects and Java objects into SOAP messages. |

# What is JAXB?

|  |  |  |  |
| --- | --- | --- | --- |
| **JAXB[[9]](#endnote-9)** | | | |
| Stands for Java Architecture for XML binding. It provides an easy way to map Java classes and XML schema hiding the complexity of XML programming. Instead of us as Java developers interacting with the abstract representation of XML up using API. So we ca simply deal with Java objects directly and the JAXB allows us to convert this XML and XML schema into Java classes and objects. | | | |
| Figure 3: Comparing JAXB to Hibernate | | | |
| **Tools** | | | |
| Starting from Java 1.6 most of these tools are a part of your JDK[[10]](#endnote-10). (for example if you go to JAVA folder on your machine and exactly to the bin folder ) | | | |
| **XJC[[11]](#endnote-11)** | It stands for XML schema compiler. It generates Java classes from a given XML Schema file. | | |
| **SCMEAGEN** | It generates XML Schema from a given set of java classes once you mark those java classes with JAXB annotations. | | |
| **Note** | | | |
| We rarely use these tools directly. We use plugins on the top these tools (like maven JAXB plugin). | | | |
| **RUNTIME API** | | It allows us to do both marshalling as well as unmarshalling. | |
| **Composition** | |
| **Marshall** | To serialize XML into Java. Marshalling is the process of converting Java objects into XML. |
| **Unmarshall** | To deserialize Java into XML. Unmarshalling is the other way around, so we convert XML back to java objects. |
| **Annotations** | Used to mark our Java classes. |
| **Tools & plugins** | | | |
| Starting of Java 1.6, most of these tools are a part of your JDK. All the tools are available in bin folder inside JDK. | | | |
| **XML Schema to JAXB Classes** | | | |
| * **Create the Project** (with Intellij IDEA or spring tool suite ...). * **Create the schemas** (We will use the existing schema files, which you would have developed from the xml and xml schema section). * **Use the JAXB Plugin**.   **Generate the stubs and use them (**we will use them to serialize and deserialize Java objects into XML**)**. | | | |

## Main structure of pom.xml

|  |  |  |
| --- | --- | --- |
| **This is the POM.xml file structure** | | |
| **Plugins** | | |
| **1** | This is the maven compiler plugin. In this plugin we are asking maven to use Java 1.8 .(You can upgrade version to 1.9 or 1.10) | |
| **2** | This is the key JAXB plugin (**version is 0.14.0**). With these plugin, we can generate stubs from XML Schema files. | |
| **Configuration element** | |
| **1** | We need to provide the Schema directory (we are using here Maven’s property project based directory. It always points to our main project directory) relative to that source/main/xsd (where our schema and configuration files are found).  Under the directory below, we are generating the stubs for patient.xsd (Patient.xsd found between schema Includes tags). We are not generating stubs for employee.xsd. |
| **2** | The file global.xjb contains xml configuration that will be used or read by the plugin. So, under the base directory of our project and exactly at src/main/resources/xsd, we find the binding file (we can have multiple binding files). These binding files tell the plugin xjcplugin or the jaxb2 plugin in how the Java code should look like when it is generated. |
| **3** | We are asking the plugin to output in this directory |
| **pom structure (build section 3)** | | |
| **If you save the pom.xml, the goal is going automatically run. Therefore, It rebuilds the project and generates the subs under the generate directory. Alternatively, we can generate them manually with the maven dependency manager.** | | |

## Customize a generated code using bindings file

|  |  |  |
| --- | --- | --- |
| **global.xjb (file structure)** | | |
| The format **.xjb** stands for xml to java binding. Using these file, we can customize the JAXB code generation process by providing this binding file to the maven plugin. | | |
| **1** | This is the root element **jaxb:bindings.** There are all the namespaces from which we can use several elements in the file. | |
| **2** | **1** | The tag xjc:simple will tell the xjc compiler that it should include the annotation Simple type on every Java class that gets generated. (It will happen automatically if we take it off) |
|  |
| **2** | You are telling the Serializable. In the generated file, you will see a serialization version ID that affects the Java serialization (we are providing -1 as default value). |
|  |
| **3** | In this section, we can even bind the xsd types to a particular java type. For exmample here, we are tellij the XJC that we want a particular type when you encounter date time in the xml schema. So we want to convert the found type to Java.util.calandar in the generated Java code. |
|  |

## Walking through a generated class (Patient.java)

### Annotation placed on a class

|  |  |  |
| --- | --- | --- |
| **Annotations placed on a class** | | |
| **Annotation** | **Definition** | |
| **@XmlRootElement** | It associates the annotated class with a root node of an XML document and tells that the root element for an object of this class when it is serialized into xml should have a name equal to “patient”. | |
| **@XmlAccessorType** | It indicates how JAXB should take in consideration the fields of a class (All the fields or properties of a class are taken into account by default in the process of generating an XML documents) except those that are annotated @XmlTransient. It is important to note that by default, if you annotate a private filed without annotating the class with @XmlAccessType.FIELD, things are likely to go wrong. Indeed JAXB will see the same field twice:  Once because of the annotation.  Second because it takes in consideration the public getter | |
| value | indication |
| .FIELD | Indicates that all non-static fields of the class are taken into account. |
| .PROPERTY | Indicates that all pairs of getters/setters are taken into account. |
| .PUBLIC | Indicates that all pairs of getters/setters and all non-static public fields will be taken into account. |
| .NONE | Indicates that no field or property is taken into account. |
| **@XmlType** | It specify the order in which these fields should be serialized when they are converted into xml. | |

### Annotation placed on a field or getter

|  |  |
| --- | --- |
| **Annotations placed on a field or a getter** | |
| **Annotation** | **Definition** |
| **@XmlElement** | Used to associate a field or a getter with a node of an XML document. It allows to specify the name of this element, its namespace, its default value …By default, the name of XML elements will be the name of the fields but we can customize that by using this annotation within the specification of name inside with validation dynamically. |
| **@XmlAttribute** | Allows the annotated field to be written in an XML attribute rather than in sub-element of parent XML element. You can set the name of this attribute, the namespace to which it belongs, and require it to be present (required attribute). |

# What is Apache CXF?

|  |  |  |  |
| --- | --- | --- | --- |
| **Web services Stack (Engine)** | | | |
| Provides us with various tools to build and run web services. This allows developers to focus on the business logic and the application itself. | | | |
| **Apache CXF** | | | |
| Apache CXF is one of the powerful and popular services engines in the java space because it implements both JAXWS and JAXRS[[12]](#endnote-12). Using Apache CXF, we can develop both the web services providers and web services consumers for both soap and Restful web services.  CXF comes with a SOAP and REST engine(it do two things):   * It serializes the XML or JSON or any other formatted messages requests and responses into Java objects and from Java objects to XML or JSON. * It dispatches the incoming requests to the appropriate web service endpoint. | | | |
|  | | | |
| **Web Service Standards** | | | |
| It implements almost all the Web services standards out there like:   * WS-Security * WS-Transaction Management * WS-Policy | | | |
| **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. | | | |
| **Configuration** | | | |
| **Spring Configuration** | | CXF uses spring to configure all the web services. | |
| **Annotations** | |  | |
| **Extends and Customize** | | | |
| Adding any customization for our application which CXF doesn’t have. | | | |
| **Interceptors** | | |  |
| **Handlers** | | |  |
| **Documentation and Samples** | | | |
| CXF comes with a lot of documentation on their website and it uses a lot of samples examples. | | | |

# Legend of acronyms

1. **SOAP:** Simple Object Access Protocol [↑](#endnote-ref-1)
2. **XML:** Extensible Markup Language [↑](#endnote-ref-2)
3. **HTTP:** Hypertext Transfer Protocol [↑](#endnote-ref-3)
4. **WSDL:** XML notation for describing a web service [↑](#endnote-ref-4)
5. **W3C:** The World Wide Web Consortium [↑](#endnote-ref-5)
6. **Apache CXF:** Apache Celtix and XFire (two projects) [↑](#endnote-ref-6)
7. **RPC:** Remote Procedure Call [↑](#endnote-ref-7)
8. **JAXWS:** Java API for XML Web Services [↑](#endnote-ref-8)
9. **JAXB:** Java Architecture for XML Binding [↑](#endnote-ref-9)
10. **JDK:** Java Developer kit [↑](#endnote-ref-10)
11. **XJC:** Java Architecture for XML Binding Compiler [↑](#endnote-ref-11)
12. **JAXRS:** JAVA API (Application Programming Interface) for Restful Web Services. [↑](#endnote-ref-12)