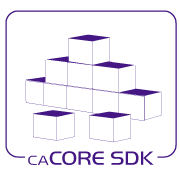
Next Generation Service Development Kit



Version: 0.1

Last Modified: 8/3/2010

Team : caCORE SDK

Client : National Cancer Institute -   
 Center for Bioinformatics and Information Technology,

National Institutes of Health,

US Department of Health and Human Services

Document History

Revision History

| **Version Number** | **Revision Date** | **Author** | **Summary of Changes** |
| --- | --- | --- | --- |
| 0.1 | 07/30/2010 | Prasad Konka | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Review

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Team/Role** | **Version** | **Date Reviewed** | **Reviewer Comments** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Related Documents

More information can be found in the following related documents:

|  |
| --- |
| **Document Name** |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Table of Contents

1. Introduction 4

2. Service Development Framework 4

2.1 Service Generation 6

**2.1.1** **CXF & JAX-WS** 9

**2.1.2** **Annotations** 10

**2.1.3** **Interceptors** 12

**2.1.4** **Error Handling** 16

**2.1.5** **Data bindings** 17

2.2 The Approach 18

2.3 Service Assembler 19

2.4 Service Synchronizer 19

# Introduction

The caCORE tools and APIs developed by the National Cancer Institute Center for Bioinformatics and Information Technology (NCI CBIIT) provide the building blocks for development of interoperable information management systems. As the Cancer Biomedical Informatics Grid (caBIG®) community continues to expand along with evolution of technologies, standards and policies – the need for SOA based service generation framework that enables connection and coordinated behavior between different participants has been essential.

This document proposes a design for next generation SOA based service generation framework following service characteristics and standards. The initial focus of this document is to propose a design to automatically generate a deployable enterprise service from a model. This document will be further expanded and extended to address all other important aspects of enterprise service like security, transaction, composition, orchestration, choreography, service registry and automated certification.

The scope of this document is limited to following items:

1. Generate various service artifacts
2. Provide pluggable business interface implementation to facilitate loose coupling between service and business implementation
3. Assemble service artifacts and business implementation to generate deployable enterprise service.
4. Update / regenerate enterprise service without effecting business implementation.

# Service Development Framework

Service-Oriented Architecture (SOA) and Enterprise Architecture are mutually reinforcing disciplines that address how IT groups can address system design and team organization to best create more re-usable and survivable systems which break down the traditional application silos. Interoperability is the most important principle of SOA. Enterprise Services enable loosely coupled, platform, operating system and programming language independent solution to generate interoperable applications. Enterprise Services generally have the following characteristics:

* Services may be individually useful, or they can be integrated—composed—to provide higher-level services. Among other benefits, this promotes re-use of existing functionality.
* Services communicate with their clients by exchanging messages: they are defined by the messages they can accept and the responses they can give.
* Services can participate in a workflow, where the order in which messages are sent and received affects the outcome of the operations performed by a service. This notion is defined as “service choreography.”
* Services may be completely self-contained, or they may depend on the availability of other services, or on the existence of a resource such as a database. In the simplest case, a service might perform a calculation such as computing the cube root of a supplied number without needing to refer to any external resource, or it may have pre-loaded all the data that it needs for its lifetime.
* Services advertise details such as their capabilities, interfaces, policies, and supported communications protocols. Implementation details such as programming language and hosting platform are of no concern to clients, and are not revealed.

Enterprise services must be agnostic regarding the choice of operating system, object model, and programming language to succeed in the heterogeneity of the Web. Enterprise services can be realized through the use of web services, as one of the key benefits of web services is interoperability, which allows different distributed web services to run on a variety of software platforms and hardware architectures.

The service generation framework should also support following important features:

* Multi-protocol service invocations: Ability to specify a different protocol using the same API, for example by changing the URL
* Federated Service Registry: Ability to register, synchronize, discover and invocate services in a federated service registry to reduce the need for centralized approach.
* Transactional: Ability to support transactional capabilities in a workflow
* Multiple Message exchange patterns: Ability to support synchronous and asynchronous service interactions
* Behavioral semantics containing semantically annotated behavioral descriptions
* Testable conformance framework to assert with specifications across different viewpoints

## Service Generation

Following diagram shows high level service generation process. A service generator is responsible to generate following service artifacts:

1. **WSDL:** It is used to describe the message syntax associated with the invocation and response of a web service. A WSDL description describes three fundamental properties of a web service:
   1. *What* a service does – The operations (methods) the service provides, and the data (arguments and returns) needed to invoke them
   2. *How* a service is accessed – Details of the data formats and protocols necessary to access the service operations
   3. *Where* the service is located – Details of the protocol-specific network address, such as a URL
2. **XSD:** It is used to describe schema of data structures used in a data exchange. This schema will be referred by the service to validate input and output types of operations.
3. **POJO:** These are the java beans generated confirming the format defined by the schema. These java beans built with standardized data types are exchanged between the interfacing services to enable interoperability.
4. **Enterprise service skeleton:** Service skeleton is the collection of necessary service artifacts to implement and deploy a service including:

* Ant processes for build, deploy, and test operations
* Standard interface for both client and service to implement
* Fully implemented client APIs
* Stub implemented service
* configuration to support service metadata and the registration of metadata and properties

1. **Behavioral semantics:** This enables standard WSDL operate at the syntactic level needed to represent the requirements and capabilities of a Web Services. This allows Web service developers to annotate their Web services with their choice of ontology language such as UML or OWL. This is significant because the ability to reuse existing domain models expressed in modeling languages like UML can greatly alleviate the need to separately model semantics. The Semantics provides a process level description of the service which, in addition to functional information, models the preconditions and postconditions of the process so that the evolution of the domain can be logically inferred. It relies on ontologies to formalize domain concepts which are shared among services.
2. **ECCF documents:**  ECCF is “a grammar for navigating and quantitatively evaluating a layered collection of artifacts that explicitly express the totality of the static and behavioral semantics necessary to achieve WI in a loosely coupled, technology-diverse, distributed deployment environment.” Following ECCF guidelines, the conceptual, platform independent and platform dependant specifications will be generated automatically driven a model.

Service generator is pluggable Ecore component following “Title of Bediako document” design specification. Service generator is a composite component orchestrating individual Ecore components responsible to generate each artifact. The flexible nature of this architecture enables adding new generators as needed without disturbing a predefined process. Conversion between each of these artifacts is also supported. For example, service generation framework would support generating WSDL from XSD or POJOs, XSD from POJOs etc.

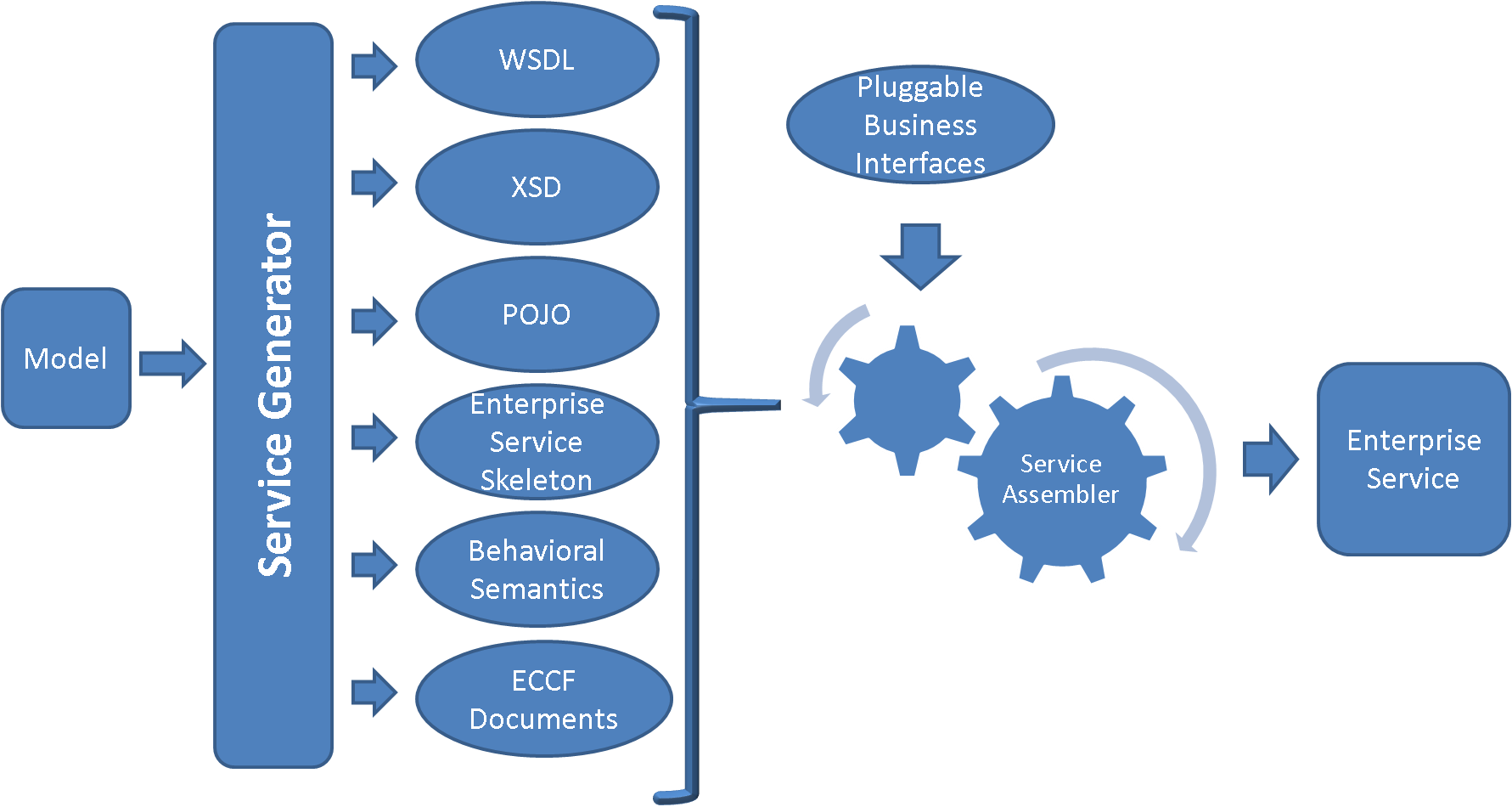


Figure : Service generation process

As shown in the diagram above, the service generator is responsible in generating service artifacts from a given information or data model. These service artifacts by themselves don’t serve business motivations unless the service implementation is coupled with business implementation. The service assembler is responsible to:

* Plug in business implementation with the service implementation. This would enable loose coupling between the service implementation and the business implementation. This is important when a service is regenerated from the model. This loose coupling makes service regeneration easy by focusing on service artifacts only.
* Generate directory structure and package all artifacts into a deployable enterprise service.

Generating source code can be powerful, but the program that writes the code can quickly become very complex and hard to understand. One way to reduce complexity and increase readability is to use templates. The Eclipse Modeling Framework (EMF) project contains two very powerful tools for generating source code: JET (Java Emitter Templates) and JMerge (Java Merge). With JET, a JSP-like syntax (actually a subset of the JSP syntax) makes it easy to write templates expressed the code in required format to generate. JET is a generic template engine that can be used to generate SQL, XML, Java source code and other output from templates. It is located in the org.eclipse.emf.codegen plug-in as part of the EMF runtime download.

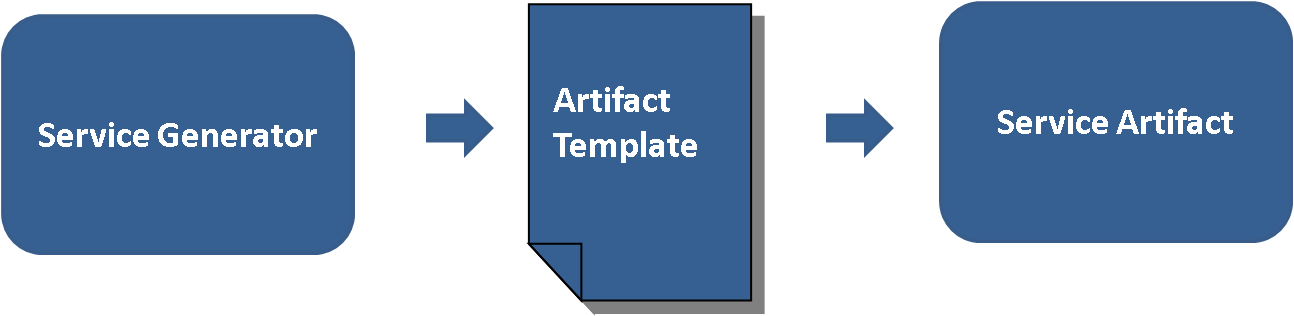


Figure : Template approach

Following is an example of JET file contents used to generate XML source code.

|  |
| --- |
| <%@ jet package="hello" imports="java.util.\*" class="XMLDemoTemplate" %>  <% List elementList = (List) argument; %>  <?xml version="1.0" encoding="UTF-8"?>  <demo>  <% for (Iterator i = elementList.iterator(); i.hasNext(); ) { %>  <element><%=i.next().toString()%></element>  <% } %>  </demo> |

The code below shows how to invoke the template instance.

|  |
| --- |
| List data = new ArrayList();  data.add("first");  data.add("second");  data.add("third");    XMLDemoTemplate generateXml = new XMLDemoTemplate();  String result = generateXml.generate(data);  System.out.println(result); |

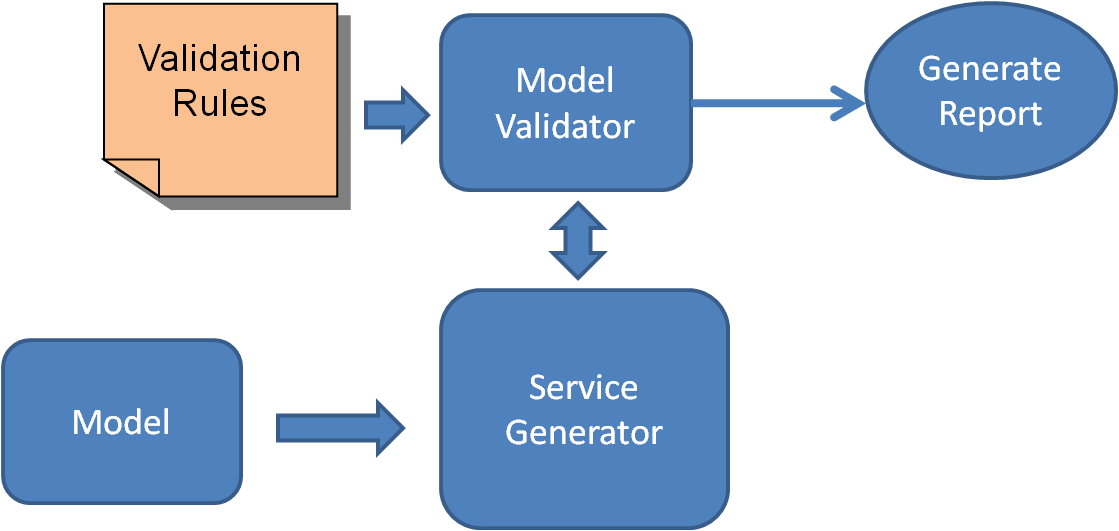
This prints the following XML result to the console:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <demo>  <element>first</element>  <element>second</element>  <element>third</element>  </demo> |

### **Validation**

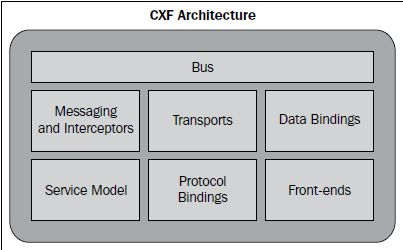
As shown and described above, model is the source of service generation. Service generator will need to validate the model before it starts generating different artifacts.

It should validate model format, model mappings and tag values. As shown in the diagram below, model validator is a separate component by itself that can be utilized by any component wish to validate a model. Model validator will support configurable validation rules through a standard XML format. This would promote loose coupling and reusability between these components. By running model validation, model validator will generate a human readable formatted output in XML and communicate back it to the caller.



### **CXF & JAX-WS**

The Apache CXF framework helps to develop web services using standards based programming model and also provides a flexible deployment model for deploying web services. Following diagram show the architecture of CXF.



Busis 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.

CXF provides the concept of frontend modeling, which allows creating web services using different frontend APIs. The APIs let users create a web service using simple factory beans and JAX-WS implementation. It also lets users create dynamic web service clients. The primary frontend supported by CXF is JAX-WS.

JAX-WS is a specification that establishes the semantics to develop, publish, and consume web services. JAX-WS simplifies web service development. JAX-WS 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 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.

CXF supports two approaches for web service development: Code-First and Contract-First. With the Code-first approach, it starts 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. With the contract-first approach, CXF provides tools to generate various service artifacts.

Service development framework will make use of open-source CXF and JAX-WS capabilities and tools to jump start with its requirements. CXF supports automated code generation of the following:

* Java to WSDL
* WSDL to Java
* XSD to WSDL
* WSDL to XML
* WSDL to SOAP
* WSDL to service

### **Annotations**

JAX-WS 2.0 uses several annotations defined by JSR-181. Annotations play a critical role in JAX-WS 2.0. First, annotations are used in mapping Java to WSDL and schema. Second, annotations are used a runtime to control how the JAX-WS runtime processes and responds to web service invocations.

Following is an example of annotating a java interface to be a service interface.

|  |
| --- |
| package com.nih.nci.sdk.demo;  @WebService  public interface HelloWorld {  String sayHi(@WebParam(name="text") String text);  } |

The @WebService annotation on the implementation class lets CXF know about the interface.

The @WebParam annotation is necessary as java interfaces do not store the Parameter name in the .class file.

Following example annotates implementation class as a service implementation.

|  |
| --- |
| package com.nih.nci.sdk.demo;  import javax.jws.WebService;  @WebService(endpointInterface = " com.nih.nci.sdk.demo.HelloWorld",  serviceName = "HelloWorld")  public class HelloWorldImpl implements HelloWorld {  public String sayHi(String text) {  System.out.println("sayHi called");  return "Hello " + text;  }  } |

Following example shows a way to publish the service.

|  |
| --- |
| HelloWorldImpl implementor = new HelloWorldImpl();  String address = "http://localhost:9000/helloWorld";  Endpoint.publish(address, implementor); |

Following example is another way to publish the service with more control over its behavior.

|  |
| --- |
| HelloWorldImpl implementor = new HelloWorldImpl();  JaxWsServerFactoryBean svrFactory = new JaxWsServerFactoryBean();  svrFactory.setServiceClass(HelloWorld.class);  svrFactory.setAddress("http://localhost:9000/helloWorld");  svrFactory.setServiceBean(implementor);  svrFactory.getInInterceptors().add(new LoggingInInterceptor());  svrFactory.getOutInterceptors().add(new LoggingOutInterceptor());  svrFactory.create(); |

Following example shows client code used to access the service.

|  |
| --- |
| public final class Client {  private static final QName SERVICE\_NAME  = new QName("http://server.nci.demo/", "HelloWorld");  private static final QName PORT\_NAME  = new QName("http://server.nci.demo/", "HelloWorldPort");  private Client() { }  public static void main(String args[]) throws Exception {  Service service = Service.create(SERVICE\_NAME);  // Endpoint Address  String endpointAddress = "http://localhost:9000/helloWorld";  // Add a port to the Service  service.addPort(PORT\_NAME, SOAPBinding.SOAP11HTTP\_BINDING, endpointAddress);    HelloWorld hw = service.getPort(HelloWorld.class);  System.out.println(hw.sayHi("World"));  }  } |

This is another client example that provides more control with interceptors.

|  |
| --- |
| JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();  factory.getInInterceptors().add(new LoggingInInterceptor());  factory.getOutInterceptors().add(new LoggingOutInterceptor());  factory.setServiceClass(HelloWorld.class);  factory.setAddress("http://localhost:9000/helloWorld");  HelloWorld client = (HelloWorld) factory.create();  String reply = client.sayHi("HI");  System.out.println("Server said: " + reply);  System.exit(0); |

All the examples shown above can be created by Service development framework automatically using CXF, JAX-WS tools and EMF templates.

### **Interceptors**

The interceptors are responsible for transforming messages between the raw data transported across the wire and the Java objects handled by the endpoint's implementation code. The interceptors are organized into phases to ensure that processing happens on the proper order. Apache CXF provides powerful runtime implementation to implement and execute interceptors. Following description is focused around CXF implementation. A similar implementation can be used in place of CXF via loosely coupled nature of this enterprise service framework. These interceptors can implement non-functional logic such as access control, logging, monitoring, etc. leaving only the core business functionality to the Web service developer

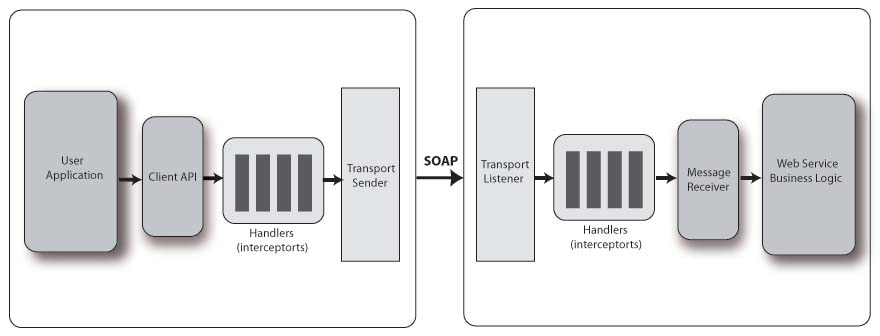


Figure : Interceptors

Interceptors are used with both CXF clients and CXF servers. When a CXF client invokes a CXF server, there is an outgoing interceptor chain for the client and an incoming chain for the server. When the server sends the response back to the client, there is an outgoing chain for the server and an incoming one for the client. Additionally, in the case of SOAPFaults, a CXF web service will create a separate outbound error handling chain and the client will create an inbound error handling chain.

Some examples of interceptors inside CXF include:

* SoapActionInterceptor - Processes the SOAPAction header and selects an operation if it's set.
* StaxInInterceptor - Creates a Stax XMLStreamReader from the transport input stream.
* Attachment(In/Out)Interceptor - Turns a multipart/related message into a series of attachments.

Following sequence diagram shows the interaction between different components invoking an interceptor.

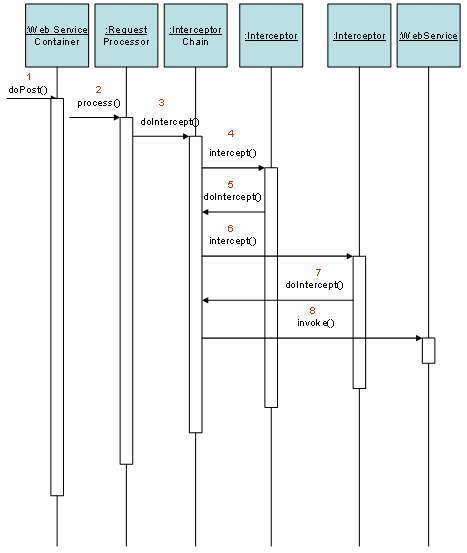


Figure : Interceptors sequence diagram

InterceptorChains are divided up into Phases. The phase that each interceptor runs in is declared in the interceptor's constructor. Each phase may contain many interceptors. On the incoming chains, you'll have the following phases:

On the incoming chains, following are the phases:

|  |  |
| --- | --- |
| **Phase** | **Functions** |
| RECEIVE | Transport level processing |
| (PRE/USER/POST)\_STREAM | Stream level processing/transformations |
| READ | This is where header reading typically occurs. |
| (PRE/USER/POST)\_PROTOCOL | Protocol processing, such as JAX-WS SOAP handlers |
| UNMARSHAL | Unmarshalling of the request |
| (PRE/USER/POST)\_LOGICAL | Processing of the umarshalled request |
| PRE\_INVOKE | Pre invocation actions |
| INVOKE | Invocation of the service |
| POST\_INVOKE | Invocation of the outgoing chain if there is one |

On the outgoing chain there are the following phases:

|  |  |
| --- | --- |
| **Phase** | **Functions** |
| SETUP | Any set up for the following phases |
| (PRE/USER/POST)\_LOGICAL | Processing of objects about to marshalled |
| PREPARE\_SEND | Opening of the connection |
| PRE\_STREAM |  |
| PRE\_PROTOCOL | Misc protocol actions. |
| WRITE | Writing of the protocol message, such as the SOAP Envelope. |
| MARSHAL | Marshalling of the objects |
| (USER/POST)\_PROTOCOL | Processing of the protocol message. |
| (USER/POST)\_STREAM | Processing of the byte level message |
| SEND | Final sending of message and closing of transport stream |

Following is an example of SOAP message interceptor that runs before/after certain other interceptors defined in the same phase.

|  |
| --- |
| public class MyInterceptor extends AbstractSoapInterceptor {  public MyInterceptor() {  super(Phase.USER\_PROTOCOL);  // MyInterceptor needs to run after SomeOtherInterceptor  getAfter().add(SomeOtherInterceptor.class.getName());  // MyInterceptor needs to run before YetAnotherInterceptor  getBefore().add(YetAnotherInterceptor.class.getName());  }  ...  } |

Interceptors can be added to a service either through annotations or CXF bus configuration. Following is an example of adding interceptors through annotations.

|  |
| --- |
| @org.apache.cxf.interceptor.InInterceptors (interceptors = {"com.example.Test1Interceptor" })  @org.apache.cxf.interceptor.InFaultInterceptors (interceptors = {"com.example.Test2Interceptor" })  @org.apache.cxf.interceptor.OutInterceptors (interceptors = {"com.example.Test1Interceptor" })  @org.apache.cxf.interceptor.InFaultInterceptors (interceptors = {"com.example.Test2Interceptor","com.example.Test3Intercetpor" })  @WebService(endpointInterface = "org.apache.cxf.javascript.fortest.SimpleDocLitBare",  targetNamespace = "uri:org.apache.cxf.javascript.fortest")  public class SayHiImplementation implements SayHi {  public long sayHi(long arg) {  return arg;  }  ...  } |

Service generation framework will read interceptor configuration from the model through tag values and generate annotated service interface code as shown above.

### **Error Handling**

An exception is any error condition or unexpected behavior encountered by an executing program. When there is an exception during the execution of the Web service, the Web service should not only capture the exceptions, but also communicate the exception back to the consumers of the Web service. Because Web services provide a platform-independent of way of leveraging a specific functionality, the exceptions that occur in the Web Services must also be communicated in a platform-independent manner. To accomplish this, SoapException class can be used that abstracts the complexities of the SOAP fault creation process. The SoapException class consists of the following properties that need to be populated before throwing the exception to the consumers.

* Message—Contents of the exception
* Code—Enum constant that specifies the type of Fault code (e.g. ClientFaultCode, ServerFaultCode)
* Actor—URL of the Web service method where the exception has occurred
* Detail—Detail element can be used to communicate more information about the exception to the callers

One can also register a custom CXF out fault interceptor which can handle all the exceptions by writing directly to the HttpServletResponse stream or XMLStreamWriter (as XMLFaultOutInterceptor does).

The WebFault annotation is used when mappingWSDL faults to Java exceptions, see section 2.5. It is used to capture the name of the fault element used when marshalling the JAXB type generated from the global element referenced by the WSDL fault message. It can also be used to customize the mapping of service specific exceptions to WSDL faults.

Following is an example of defining an exception.

|  |
| --- |
| package com.example.customerservice;  @WebFault(name="NoSuchCustomer")  @XmlAccessorType( XmlAccessType.FIELD )  public class NoSuchCustomerException extends RuntimeException {  /\*\*  \* We only define the fault details here. Additionally each fault has a message  \* that should not be defined separately  \*/  String customerName;  } |

Following is an example of using the exception.

|  |
| --- |
| package com.example.customerservice;  @WebService  public interface CustomerService {  public Customer[] getCustomersByName(@WebParam(name="name") String name) throws NoSuchCustomerException;  } |

### **Data bindings**

Data binding components are responsible for mapping between XML on the wire and Java objects. Each data binding implements a particular discipline for mapping, such as JAXB or XML Beans.

There are three parts to a data binding:

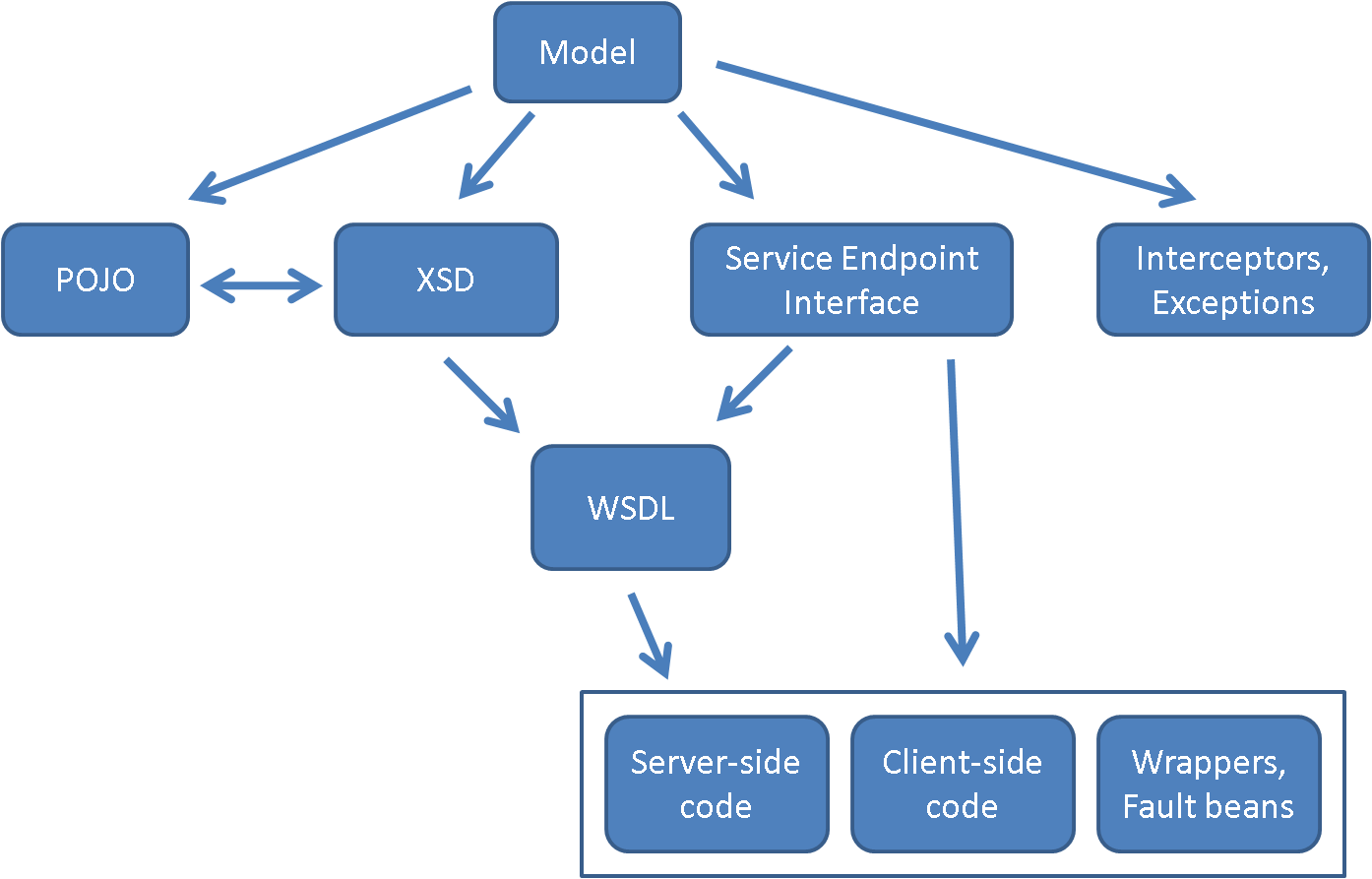
* Mapping the live data as it comes into and out of services.
* Providing XML schema based on Java objects for dynamic wsdl URLs and java2ws.
* Generating Java code from WSDL for wsdl2java (and, theoretically, dynamic clients).

All data bindings provide the live data mapping. Each data binding supports one or more formats for the data in transit. By default, CXF supports data binding by JAXB. The architecture of CXF supports pluggable nature of data binding framework to replace its default easily.

## The Approach

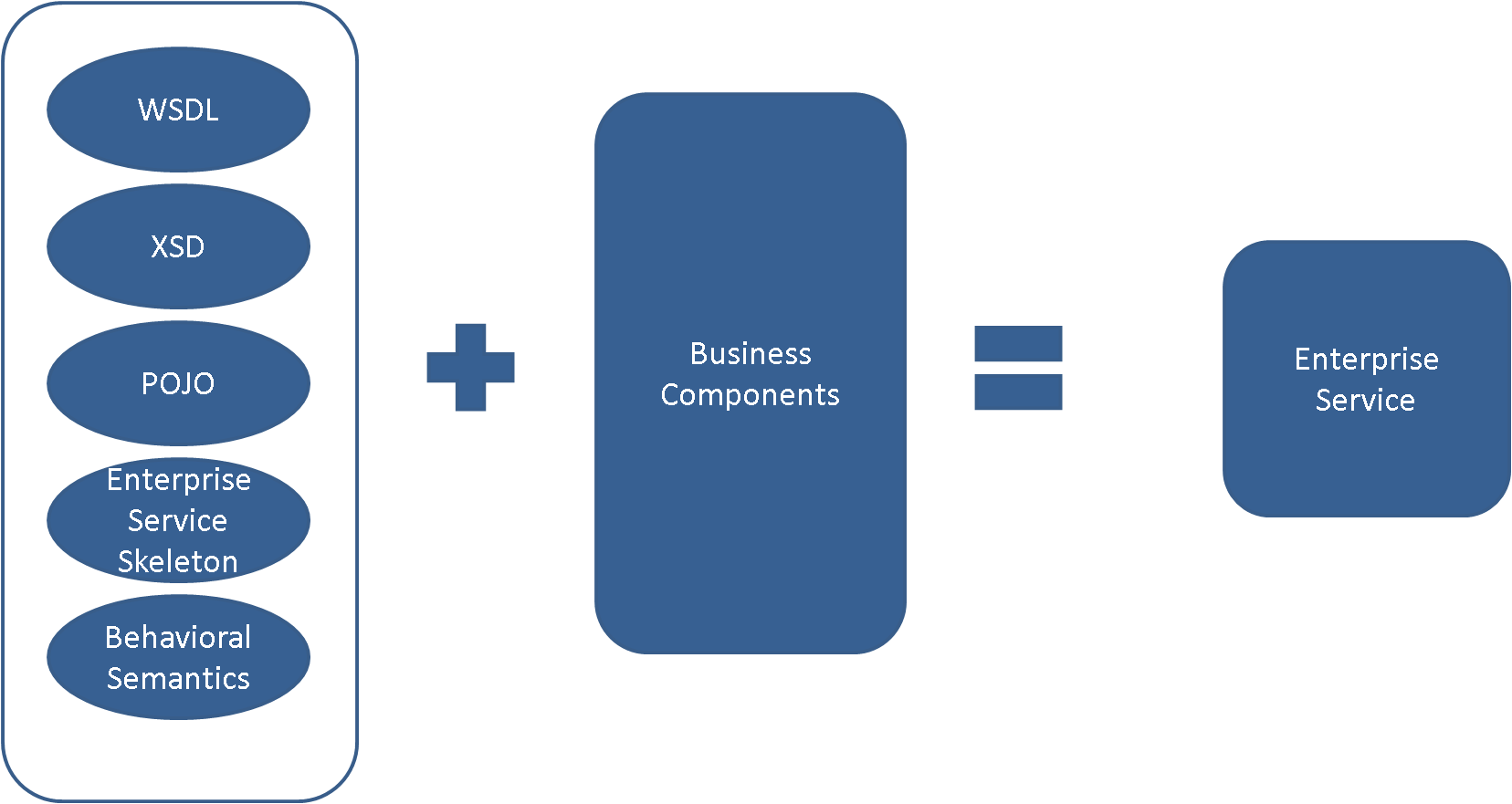
Following diagrams shows the overall approach of artifact generation. With the loose coupling nature service generation framework coupled with CXF, an enterprise deployable service can be generated from a model in two approaches.

* 1. Code first: With this approach, service generator component will generate a service endpoint interface with all standard, user opted annotations from the model. Service development framework will use CXF tools to generate WSDL or service artifacts including interceptors, exceptions, service implementation, stubs, client-side code, wrappers, fault handlers. This would also generate configuration files necessary to deploy the service.
  2. Contract first: With this approach, service generator component will generate XSD from the model. Service development framework will use CXF tools to generate WSDL and to further generate all above mentioned artifacts.



## Service Assembler

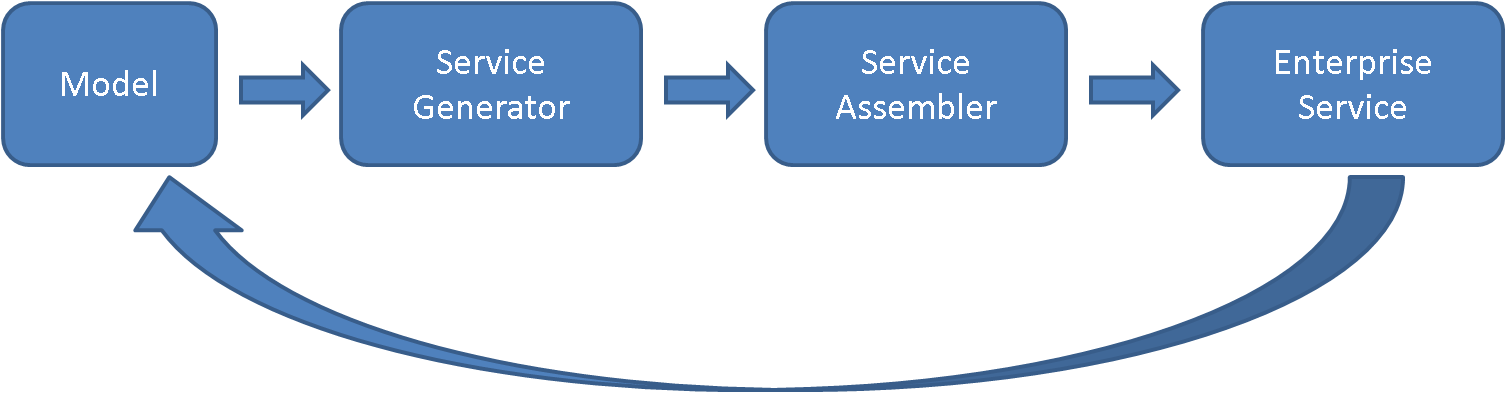
When designing an application, developers develop a logical model of what an enterprise does in terms of business objects (such as product, customer, order, bill, etc) and the services the business requires from these business objects (what is the stock level, what is the delivery schedule and so on). The developer may implement these concepts as a blend of service interfaces and components (the business objects). Components are normally used to implement (realize) the service functionality. It is important to distinguish between these two elements to enable loose coupling and separation of responsibilities.



Service assembler is responsible to automatically plugin service implementation with business components serving the actual business functions either through configuration or UI.

## Service Synchronizer

During development, it is often possible for a developer to update automatically generated code manually for various reasons. This could generate a gap between the model that was used to generate the code and manual modifications. It is important to keep these parts in sync to make sure any further runs to generate the service artifacts would not overwrite manual changes. Service synchronizer is responsible to capture any changes in the service artifacts and carry those changes back to the model to synchronize it with the code. Following diagram shows high level flow of the process.



As it is practical to make many types of changes to the generated artifacts, the initial approach of this design to restrict synchronization process to capture changes made to POJO operations. That is, if there are any new operations added, or changes to existing operations signature will be captured and synchronized back with the model. The synchronization process can be broken down into following components:

* Reader: This component is responsible to read generated artifacts and represent them in model format.
* Synchronizer: This component is responsible to compare original model with generated model from the updated artifacts. Synchronizer will present comparison results in a text format or on a UI with side-by-side comparison windows to move changes. Any confirmed changes will be written into the original model.