Interceptors

Interceptors and Phases

Interceptors are the fundamental processing unit inside CXF. When a service is invoked, an InterceptorChain is created and invoked. Each interceptor gets a chance to do what they want with the message. This can include reading it, transforming it, processing headers, validating the message, etc.

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.

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:

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	

After the SEND phase, there are a bunch of "*_ENDING" phases that are symmetrical to the above phases to allow the interceptors to cleanup and close anything that they had opened or started in the above phases:

Phase	Functions
SEND_ENDING	
POST_STREAM_ENDING	
USER_STREAM_ENDING	

POST_PROTOCOL_ENDING	
USER_PROTOCOL_ENDING	
MARSHAL_ENDING	
WRITE_ENDING	
PRE_PROTOCOL_ENDING	
PRE_STREAM_ENDING	
PREPARE_SEND_ENDING	
POST_LOGICAL_ENDING	
USER_LOGICAL_ENDING	
PRE_LOGICAL_ENDING	
SETUP_ENDING	Usually results in all the streams being closed and the final data being sent on the wire.

InterceptorProviders

Several different components inside CXF may provide interceptors to an InterceptorChain. These implement the InterceptorProvider interface:

```
public interface InterceptorProvider {

    List<Interceptor> getInInterceptors();

    List<Interceptor> getOutInterceptors();

    List<Interceptor> getOutFaultInterceptors();

    List<Interceptor> getInFaultInterceptors();
}
```

To add an interceptor to an interceptor chain, you'll want to add it to one of the Interceptor Providers.

```
MyInterceptor interceptor = new MyInterceptor();
provider.getInInterceptors().add(interceptor);
```

Some InterceptorProviders inside CXF are:

- Client
- Endpoint
- Service
- Bus
- Binding

Writing and configuring an Interceptor

The CXF distribution is shipped with a demo called configuration_interceptor which shows how to develop a user interceptor and configure the interceptor into its interceptor chain.

Writing an Interceptor

Writing an interceptor is relatively simple. Your interceptor needs to extend from either the AbstractPhaseInterceptor or one of its many subclasses such as AbstractSoapInterceptor. Extending from AbstractPhaseInterceptor allows your interceptor to access the methods of the Message interface. For example, AttachmentInInterceptor is used in CXF to turn a multipart/related message into a series of attachments. It looks like below:

```
import java.io.IOException;

import org.apache.cxf.attachment.AttachmentDeserializer;
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class AttachmentInInterceptor extends AbstractPhaseInterceptor<Message> {
    public AttachmentInInterceptor() {
        super(Phase.RECEIVE);
    }

    public void handleMessage(Message message) {
        String contentType = (String) message.get(Message.CONTENT_TYPE);
        if (contentType != null && contentType.toLowerCase().indexOf("multipart/related") != -1) {
            AttachmentDeserializer ad = new AttachmentDeserializer(message);
            try {
                ad.initializeAttachments();
            } catch (IOException e) {
                throw new Fault(e);
            }
        }
    }

    public void handleFault(Message messageParam) {
    }
}
```

Extending from sub-classes of AbstractPhaseInterceptor allows your interceptor to access more specific information than those in the Message interface. One of the sub-classes of AbstractPhaseInterceptor is AbstractSoapInterceptor. Extending from this class allows your interceptor to access the SOAP header and version information of the SoapMessage class. For example, SoapActionInInterceptor is used in CXF to parse the SOAP action, as a simplified version of it shows below:

```
import java.util.Collection;
import java.util.List;
import java.util.Map;

import org.apache.cxf.binding.soap.Soap11;
import org.apache.cxf.binding.soap.Soap12;
import org.apache.cxf.binding.soap.SoapMessage;
import org.apache.cxf.binding.soap.model.SoapOperationInfo;
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.helpers.CastUtils;
import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.message.Exchange;
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.Phase;
import org.apache.cxf.service.model.BindingOperationInfo;
import org.apache.cxf.service.model.OperationInfo;

public class SoapActionInInterceptor extends AbstractSoapInterceptor {

    public SoapActionInInterceptor() {
        super(Phase.READ);
        addAfter(ReadHeadersInterceptor.class.getName());
        addAfter(EndpointSelectionInterceptor.class.getName());
    }

    public void handleMessage(SoapMessage message) throws Fault {
        if (message.getVersion() instanceof Soap11) {
            Map<String, List<String>> headers = CastUtils.cast((Map)message.get(Message.PROTOCOL_HEADERS));
            if (headers != null) {
                List<String> sa = headers.get("SOAPAction");
                if (sa != null && sa.size() > 0) {
                    String action = sa.get(0);
                    if (action.startsWith("\"")) {
                        action = action.substring(1, action.length() - 1);
                    }
                    getAndSetOperation(message, action);
```

```
                }
            }
        } else if (message.getVersion() instanceof Soap12) {
          ...........
        }
    }

    private void getAndSetOperation(SoapMessage message, String action) {
        if ("".equals(action)) {
            return;
        }

        Exchange ex = message.getExchange();
        Endpoint ep = ex.get(Endpoint.class);

        BindingOperationInfo bindingOp = null;

        Collection<BindingOperationInfo> bops = ep.getBinding().getBindingInfo().getOperations();
        for (BindingOperationInfo boi : bops) {
            SoapOperationInfo soi = (SoapOperationInfo) boi.getExtensor(SoapOperationInfo.class);
            if (soi != null && soi.getAction().equals(action)) {
                if (bindingOp != null) {
                    //more than one op with the same action, will need to parse normally
                    return;
                }
                bindingOp = boi;
            }
        }
        if (bindingOp != null) {
            ex.put(BindingOperationInfo.class, bindingOp);
            ex.put(OperationInfo.class, bindingOp.getOperationInfo());
        }
    }

}
```

Note that you will need to specify the phase that the interceptor will be included in. This is done in the interceptor's constructor:

```
public class MyInterceptor extends AbstractSoapInterceptor {
  public MyInterceptor() {
    super(Phase.USER_PROTOCOL);
  }
  ...
}
```

You can also express that you would like the interceptor to run 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());
  }
  ...
}
```

You can add your interceptors into the interceptor chain either programmatically or through configuration.

Adding interceptors programmatically

To add this to your server, you'll want to get access to the Server object (see here for more info):

```
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;
...

MyInterceptor myInterceptor = new MyInterceptor();

Server server = serverFactoryBean.create();
server.getEndpoint().getInInterceptor().add(myInterceptor);
```

On the Client side the process is very similar:

```
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.frontend.ClientProxy;
...

MyInterceptor myInterceptor = new MyInterceptor();
FooService client = ... ; // created from ClientProxyFactoryBean or generated JAX-WS client

//You could also call clientProxyFactroyBean.getInInterceptor().add(myInterceptor) to add the interceptor

Client cxfClient = ClientProxy.getClient(client);
cxfClient.getInInterceptors().add(myInterceptor);

// then you can call the service
client.doSomething();
```

You can also use annotation to add the interceptors from the SEI or service class. When CXF create the server or client, CXF will add the interceptor according with the annotation.

```
@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;
   }
   ...
}
```

Adding interceptors through configuration

The configuration file page provides examples on using configuration files to add interceptors.

Adding MyInterceptor to the bus:

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

    <bean id="MyInterceptor" class="demo.interceptor.MyInterceptor"/>

    <!-- We are adding the interceptors to the bus as we will have only one endpoint/service/bus. -->

    <cxf:bus>
        <cxf:inInterceptors>
            <ref bean="MyInterceptor"/>
        </cxf:inInterceptors>
        <cxf:outInterceptors>
            <ref bean="MyInterceptor"/>
       </cxf:outInterceptors>
    </cxf:bus>
</beans>
```

For embedded Jetty-based web services, the configuration file can be declared by starting the service with the -Dcxf.config.file=server.xml option. See the server configuration section on the configuration file page for information on specifying the file for servlet WAR file-based web service implementations.

Adding MyInterceptor to your client:

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

    <http:conduit name="{http://apache.org/hello_world_soap_http}SoapPort9001.http-conduit">
      <http:client DecoupledEndpoint="http://localhost:9990/decoupled_endpoint"/>
    </http:conduit>

    <bean id="MyInterceptor" class="demo.interceptor.MyInterceptor"/>

    <!-- We are adding the interceptors to the bus as we will have only one endpoint/service/bus. -->

    <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
        <property name="inInterceptors">
            <ref bean="MyInterceptor"/>
        </property>
        <property name="outInterceptors">
            <ref bean="MyInterceptor"/>
        </property>
    </bean>
</beans>
```

To specify the client-side configuration file, start your client using the -Dcxf.config.file=client.xml option.

CXF contributed interceptors

In CXF, all the functionality of processing messages is done via interceptors. Thus, when debugging a message flow, you will come across a bunch of interceptors in the chain. Here is a list of some of the common interceptors and the functionality they provide. The source code for these interceptors is available on github.

Default JAX-WS Incoming interceptor chain (Server):

- **AttachmentInInterceptor** Parse the mime headers for mime boundaries, finds the "root" part and resets the input stream to it, and stores the other parts in a collection of Attachments
- **StaxInInterceptor** Creates an XMLStreamReader from the transport InputStream on the Message
- **ReadHeadersInterceptor** Parses the SOAP headers and stores them on the Message

- **SoapActionInInterceptor** Parses "soapaction" header and looks up the operation if a unique operation can be found for that action.
- **MustUnderstandInterceptor** Checks the MustUnderstand headers, its applicability and process it, if required
- **SOAPHandlerInterceptor** SOAP Handler as per JAX-WS
- **LogicalHandlerInInterceptor** Logical Handler as per JAX-WS
- **CheckFaultInterceptor** Checks for fault, if present aborts interceptor chain and invokes fault handler chain
- **URIMappingInterceptor** (for CXF versions <= 2.x) Can handle HTTP GET, extracts operation info and sets the same in the Message
- **DocLiteralInInterceptor** Examines the first element in the SOAP body to determine the appropriate Operation (if soapAction did not find one) and calls the Databinding to read in the data.
- **SoapHeaderInterceptor** Perform databinding of the SOAP headers for headers that are mapped to parameters
- **WrapperClassInInterceptor** For wrapped doc/lit, the DocLiteralInInterceptor probably read in a single JAXB bean. This interceptor pulls the individual parts out of that bean to construct the Object[] needed to invoke the service.
- **SwAInInterceptor** For Soap w/ Attachments, finds the appropriate attachments and assigns them to the correct spot in the parameter list.
- **HolderInInterceptor** For OUT and IN/OUT parameters, JAX-WS needs to create Holder objects. This interceptor creates the Holders and puts them in the parameter list.
- **ServiceInvokerInInterceptor** Actually invokes the service.

Default Outgoing chain stack (Server):

- **HolderOutInterceptor** For OUT and IN/OUT params, pulls the values out of the JAX-WS Holder objects (created in HolderInInterceptor) and adds them to the param list for the out message.
- **SwAOutInterceptor** For OUT parts that are Soap attachments, pulls them from the list and holds them for later.
- **WrapperClassOutInterceptor** For doc/lit wrapped, takes the remaining parts and creates a wrapper JAXB bean to represent the whole message.
- **SoapHeaderOutFilterInterceptor** Removes inbound marked headers
- **SoapActionOutInterceptor** Sets the SOAP Action
- **MessageSenderInterceptor** Calls back to the Destination object to have it setup the output streams, headers, etc... to prepare the outgoing transport.
- **SoapPreProtocolOutInterceptor** This interceptor is responsible for setting up the SOAP version and header, so that this is available to any pre-protocol interceptors that require these to be available.
- **AttachmentOutInterceptor** If this service uses attachments (either SwA or if MTOM is enabled), it sets up the Attachment marshallers and the mime stuff that is needed.
- **StaxOutInterceptor** Creates an XMLStreamWriter from the OutputStream on the Message.
- **SoapHandlerInterceptor** JAX-WS SOAPHandler
- **SoapOutInterceptor** Writes start element for soap:envelope and complete elements for other header blocks in the message. Adds start element for soap:body too.
- **LogicalHandlerOutInterceptor** JAX-WS Logical handler stuff
- **WrapperOutInterceptor** If wrapped doc/lit and not using a wrapper bean or if RPC lit, outputs the wrapper element to the stream.
- **BareOutInterceptor** Uses the databinding to write the params out.
- **SoapOutInterceptor$SoapOutEndingInterceptor** Closes the soap:body and soap:envelope
- **StaxOutInterceptor$StaxOutEndingInterceptor** Flushes the stax stream.
- **MessageSenderInt$MessageSenderEnding** Closes the exchange, lets the transport know everything is done and should be flushed to the client.