Fuse ESB

Developing Apache CXF Interceptors

Version 4.4.1
Sept. 2011

Developing Apache CXF Interceptors

Version 4.4.1

Updated: 06 Jun 2013

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux `<mwp1@cornell.edu>`

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Interceptors in the Fuse Services Framework Runtime

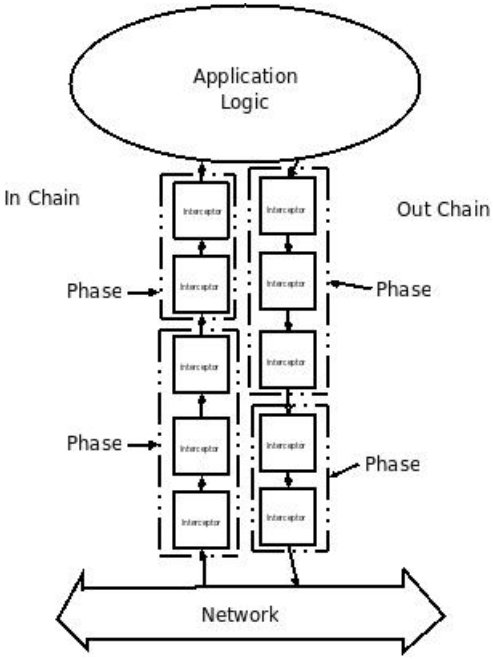
Most of the functionality in the Fuse Services Framework runtime is implemented by interceptors. Every endpoint created by the Fuse Services Framework runtime has three potential interceptor chains for processing messages. The interceptors in the these chains are responsible for transforming messages 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.

Overview

A large part of what Fuse Services Framework does entails processing messages. When a consumer makes a invocation on a remote service the runtime needs to marshal the data into a message the service can consume and place it on the wire. The service provider must unmarshal the message, execute its business logic, and marshal the response into the appropriate message format. The consumer must then unmarshal the response message, correlate it to the proper request, and pass it back to the consumer's application code. In addition to the basic marshaling and unmarshaling, the Fuse Services Framework runtime may do a number of other things with the message data. For example, if WS-RM is activated, the runtime must process the message chunks and acknowledgement messages before marshaling and unmarshaling the message. If security is activated, the runtime must validate the message's credentials as part of the message processing sequence.

Figure 1.1 on page 12 shows the basic path that a request message takes when it is received by a service provider.

Figure 1.1. Fuse Services Framework interceptor chains



Message processing in Fuse Services Framework

When a Fuse Services Framework developed consumer invokes a remote service the following message processing sequence is started:

1. The Fuse Services Framework runtime creates an outbound interceptor chain to process the request.
2. If the invocation starts a two-way message exchange, the runtime creates an inbound interceptor chain and a fault processing interceptor chain.
3. The request message is passed sequentially through the outbound interceptor chain.

   Each interceptor in the chain performs some processing on the message. For example, the Fuse Services Framework supplied SOAP interceptors package the message in a SOAP envelope.
4. If any of the interceptors on the outbound chain create an error condition the chain is unwound and control is returned to the application level code.

An interceptor chain is unwound by calling the fault processing method on all of the previously invoked interceptors.

5. The request is dispatched to the appropriate service provider.
6. When the response is received, it is passed sequentially through the inbound interceptor chain.

> **Note**
>
> If the response is an error message, it is passed into the fault processing interceptor chain.

7. If any of the interceptors on the inbound chain create an error condition, the chain is unwound.
8. When the message reaches the end of the inbound interceptor chain, it is passed back to the application code.

When a Fuse Services Framework developed service provider receives a request from a consumer, a similar process takes place:

1. The Fuse Services Framework runtime creates an inbound interceptor chain to process the request message.
2. If the request is part of a two-way message exchange, the runtime also creates an outbound interceptor chain and a fault processing interceptor chain.
3. The request is passed sequentially through the inbound interceptor chain.
4. If any of the interceptors on the inbound chain create an error condition, the chain is unwound and a fault is dispatched to the consumer.

   An interceptor chain is unwound by calling the fault processing method on all of the previously invoked interceptors.
5. When the request reaches the end of the inbound interceptor chain, it is passed to the service implementation.
6. When the response is ready it is passed sequentially through the outbound interceptor chain.

Note

If the response is an exception, it is passed through the fault processing interceptor chain.

7. If any of the interceptors on the outbound chain create an error condition, the chain is unwound and a fault message is dispatched.
8. Once the request reaches the end of the outbound chain, it is dispatched to the consumer.

Interceptors

All of the message processing in the Fuse Services Framework runtime is done by *interceptors*. Interceptors are POJOs that have access to the message data before it is passed to the application layer. They can do a number of things including: transforming the message, stripping headers off of the message, or validating the message data. For example, an interceptor could read the security headers off of a message, validate the credentials against an external security service, and decide if message processing can continue.

The message data available to an interceptor is determined by several factors:

- the interceptor's chain
- the interceptor's phase
- the other interceptors that occur earlier in the chain

Phases

Interceptors are organized into *phases*. A phase is a logical grouping of interceptors with common functionality. Each phase is responsible for a specific type of message processing. For example, interceptors that process the marshaled Java objects that are passed to the application layer would all occur in the same phase.

Interceptor chains

Phases are aggregated into *interceptor chains*. An interceptor chain is a list of interceptor phases that are ordered based on whether messages are inbound or outbound.

Each endpoint created using Fuse Services Framework has three interceptor chains:

- a chain for inbound messages

- a chain for outbound messages
- a chain for error messages

Interceptor chains are primarily constructed based on the choose of binding and transport used by the endpoint. Adding other runtime features, such as security or logging, also add interceptors to the chains. Developers can also add custom interceptors to a chain using configuration.

Developing interceptors

Developing an interceptor, regardless of its functionality, always follows the same basic procedure:

1. Determine which abstract interceptor class to extend.

   Fuse Services Framework provides a number of abstract interceptors to make it easier to develop custom interceptors.

2. Determine the phase in which the interceptor will run.

   Interceptors require certain parts of a message to be available and require the data to be in a certain format. The contents of the message and the format of the data is partially determined by an interceptor's phase.

3. Determine if there are any other interceptors that must be executed either before or after the interceptor.

   In general, the ordering of interceptors within a phase is not important. However, in certain situations it may be important to ensure that an interceptor is executed before, or after, other interceptors in the same phase.

4. Implement the interceptor's message processing logic.

5. Implement the interceptor's fault processing logic.

   If an error occurs in the active interceptor chain after the interceptor has executed, its fault processing logic is invoked.

6. Attach the interceptor to one of the endpoint's interceptor chains.

Chapter 2. The Interceptor APIs

Interceptors implement the `PhaseInterceptor` interface which extends the base `Interceptor` interface. This interface defines a number of methods used by the Fuse Services Framework's runtime to control interceptor execution and are not appropriate for application developers to implement. To simplify interceptor development, Fuse Services Framework provides a number of abstract interceptor implementations that can be extended.

Interfaces

All of the interceptors in Fuse Services Framework implement the base `Interceptor` interface shown in Example 2.1 on page 17.

Example 2.1. Base interceptor interface

```
package org.apache.cxf.interceptor;

public interface Interceptor<T extends Message>
{

  void handleMessage(T message) throws Fault;

  void handleFault(T message);

}
```

The `Interceptor` interface defines the two methods that a developer needs to implement for a custom interceptor:

handleMessage()
: The `handleMessage()` method does most of the work in an interceptor. It is called on each interceptor in a message chain and receives the contents of the message being processed. Developers implement the message processing logic of the interceptor in this method. For detailed information about implementing the `handleMessage()` method, see "Processing messages" on page 28.

handleFault()
: The `handleFault()` method is called on an interceptor when normal message processing has been interrupted. The runtime calls the `handleFault()` method of each invoked interceptor in reverse order as it unwinds an interceptor chain. For detailed information about implementing the `handleFault()` method, see "Unwinding after an error" on page 31.

Most interceptors do not directly implement the `Interceptor` interface. Instead, they implement the `PhaseInterceptor` interface shown in Example 2.2 on page 18. The `PhaseInterceptor` interface adds four methods that allow an interceptor the participate in interceptor chains.

Example 2.2. The phase interceptor interface

```
package org.apache.cxf.phase;
...

public interface PhaseInterceptor<T extends Message> extends Interceptor<T>
{

    Set<String> getAfter();

    Set<String> getBefore();

    String getId();

    String getPhase();

}
```

Abstract interceptor class

Instead of directly implementing the `PhaseInterceptor` interface, developers should extend the `AbstractPhaseInterceptor` class. This abstract class provides implementations for the phase management methods of the `PhaseInterceptor` interface. The `AbstractPhaseInterceptor` class also provides a default implementation of the `handleFault()` method.

Developers need to provide an implementation of the `handleMessage()` method. They can also provide a different implementation for the `handleFault()` method. The developer-provided implementations can manipulate the message data using the methods provided by the generic `org.apache.cxf.message.Message` interface.

For applications that work with SOAP messages, Fuse Services Framework provides an `AbstractSoapInterceptor` class. Extending this class provides the `handleMessage()` method and the `handleFault()` method with access to the message data as an `org.apache.cxf.binding.soap.SoapMessage` object. `SoapMessage` objects have methods for retrieving the SOAP headers, the SOAP envelope, and other SOAP metadata from the message.

Chapter 3. Determining When the Interceptor is Invoked

Interceptors are organized into phases. The phase in which an interceptor runs determines what portions of the message data it can access. An interceptor can determine its location in relationship to the other interceptors in the same phase. The interceptor's phase and its location within the phase are set as part of the interceptor's constructor logic.

When developing a custom interceptor, the first thing to consider is where in the message processing chain the interceptor belongs. The developer can control an interceptor's position in the message processing chain in one of two ways:

- Specifying the interceptor's phase
- Specifying constraints on the location of the interceptor within the phase

Typically, the code specifying an interceptor's location is placed in the interceptor's constructor. This makes it possible for the runtime to instantiate the interceptor and put in the proper place in the interceptor chain without any explicit action in the application level code.

Specifying an interceptor's phase

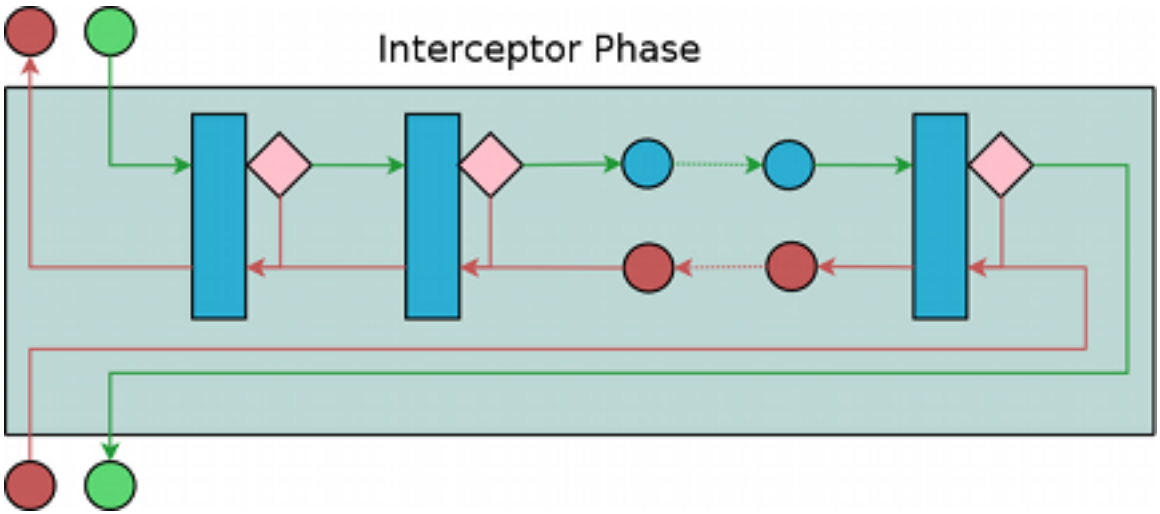
Overview

Interceptors are organized into phases. An interceptor's phase determines when in the message processing sequence it is called. Developers specify an interceptor's phase its constructor. Phases are specified using constant values provided by the framework.

Phase

Phases are a logical collection of interceptors. As shown in Figure 3.1 on page 20, the interceptors within a phase are called sequentially.

Figure 3.1. An interceptor phase



The phases are linked together in an ordered list to form an interceptor chain and provide defined logical steps in the message processing procedure. For example, a group of interceptors in the `RECEIVE` phase of an inbound interceptor chain processes transport level details using the raw message data picked up from the wire.

There is, however, no enforcement of what can be done in any of the phases. It is recommended that interceptors within a phase adhere to tasks that are in the spirit of the phase.

The complete list of phases defined by Fuse Services Framework can be found in Appendix A on page 51.

Specifying a phase

Fuse Services Framework provides the `org.apache.cxf.Phase` class to use for specifying a phase. The class is a collection of constants. Each phase defined by Fuse Services Framework has a corresponding constant in the `Phase` class. For example, the `RECEIVE` phase is specified by the value `Phase.RECEIVE`.

Setting the phase

An interceptor's phase is set in the interceptor's constructor. The `AbstractPhaseInterceptor` class defines three constructors for instantiating an interceptor:

- `public AbstractPhaseInterceptor(String phase)`—sets the phase of the interceptor to the specified phase and automatically sets the interceptor's id to the interceptor's class name.

Tip

This constructor will satisfy most use cases.

- `public AbstractPhaseInterceptor(String id, String phase)`—sets the interceptor's id to the string passed in as the first parameter and the interceptor's phase to the second string.
- `public AbstractPhaseInterceptor(String phase, boolean uniqueId)`—specifies if the interceptor should use a unique, system generated id. If the *`uniqueId`* parameter is `true`, the interceptor's id will be calculated by the system. If the *`uniqueId`* parameter is `false` the interceptor's id is set to the interceptor's class name.

The recommended way to set a custom interceptor's phase is to pass the phase to the `AbstractPhaseInterceptor` constructor using the `super()` method as shown in Example 3.1 on page 21.

Example 3.1. Setting an interceptor's phase

```
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
 {

    public StreamInterceptor()
   {
     super(Phase.PRE_STREAM);
```

```
    }
}
```

The `StreamInterceptor` interceptor shown in Example 3.1 on page 21 is placed into the `PRE_STREAM` phase.

Constraining an interceptors placement in a phase

Overview

Placing an interceptor into a phase may not provide fine enough control over its placement to ensure that the interceptor works properly. For example, if an interceptor needed to inspect the SOAP headers of a message using the SAAJ APIs, it would need to run after the interceptor that converts the message into a SAAJ object. There may also be cases where one interceptor consumes a part of the message needed by another interceptor. In these cases, a developer can supply a list of interceptors that must be executed before their interceptor. A developer can also supply a list of interceptors that must be executed after their interceptor.

Important

The runtime can only honor these lists within the interceptor's phase. If a developer places an interceptor from an earlier phase in the list of interceptors that must execute after the current phase, the runtime will ignore the request.

Add to the chain before

One issue that arises when developing an interceptor is that the data required by the interceptor is not always present. This can occur when one interceptor in the chain consumes message data required by a later interceptor. Developers can control what a custom interceptor consumes and possibly fix the problem by modifying their interceptors. However, this is not always possible because a number of interceptors are used by Fuse Services Framework and a developer cannot modify them.

An alternative solution is to ensure that a custom interceptor is placed before any interceptors that will consume the message data the custom interceptor requires. The easiest way to do that would be to place it in an earlier phase, but that is not always possible. For cases where an interceptor needs to be placed before one or more other interceptors the Fuse Services Framework's `AbstractPhaseInterceptor` class provides two `addBefore()` methods.

As shown in Example 3.2 on page 23, one takes a single interceptor id and the other takes a collection of interceptor ids. You can make multiple calls to continue adding interceptors to the list.

Example 3.2. Methods for adding an interceptor before other interceptors

```
public void addBefore(String i);

public void addBefore(Collection<String> i);
```

As shown in Example 3.3 on page 24, a developer calls the `addBefore()` method in the constuctor of a custom interceptor.

Example 3.3. Specifying a list of interceptors that must run after the current interceptor

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addBefore(HolderOutInterceptor.class.getName());
    }

...

}
```

Tip

Most interceptors use their class name for an interceptor id.

Add to the chain after

Another reason the data required by the interceptor is not present is that the data has not been placed in the message object. For example, an interceptor may want to work with the message data as a SOAP message, but it will not work if it is placed in the chain before the message is turned into a SOAP message. Developers can control what a custom interceptor consumes and possibly fix the problem by modifying their interceptors. However, this is not always possible because a number of interceptors are used by Fuse Services Framework and a developer cannot modify them.

An alternative solution is to ensure that a custom interceptor is placed after the interceptor, or interceptors, that generate the message data the custom interceptor requires. The easiest way to do that would be to place it in a later phase, but that is not always possible. The `AbstractPhaseInterceptor` class provides two `addAfter()` methods for cases where an interceptor needs to be placed after one or more other interceptors.

As shown in Example 3.4 on page 24, one method takes a single interceptor id and the other takes a collection of interceptor ids. You can make multiple calls to continue adding interceptors to the list.

Example 3.4. Methods for adding an interceptor after other interceptors

```
public void addAfter(String i);

public void addAfter(Collection<String> i);
```

As shown in Example 3.5 on page 25, a developer calls the `addAfter()` method in the constuctor of a custom interceptor.

Example 3.5. Specifying a list of interceptors that must run before the current interceptor

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addAfter(StartingOutInterceptor.class.getName());
    }

...

}
```

Tip

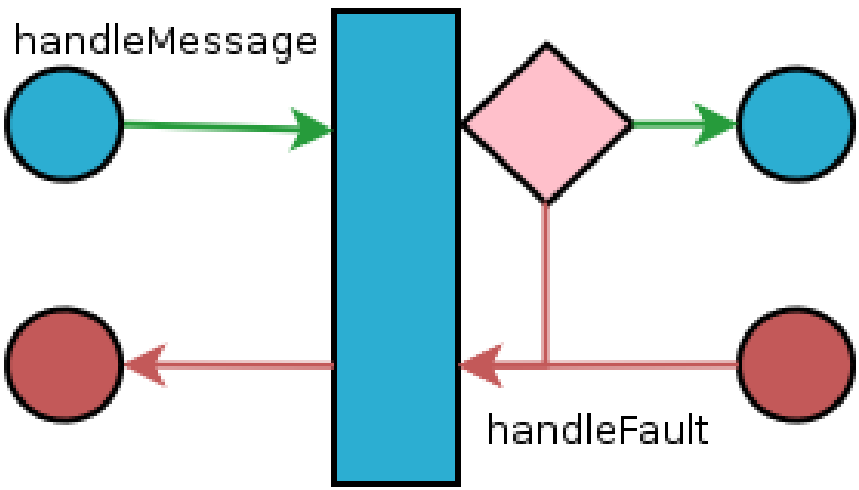
Most interceptors use their class name for an interceptor id.

Chapter 4. Implementing the Interceptors Processing Logic

Interceptors are straightforward to implement. The bulk of their processing logic is in the `handleMessage()` method. This method receives the message data and manipulates it as needed. Developers may also want to add some special logic to handle fault processing cases.

Figure 4.1 on page 27 shows the process flow through an interceptor.

Figure 4.1. Flow through an interceptor



In normal message processing, only the `handleMessage()` method is called. The `handleMessage()` method is where the interceptor's message processing logic is placed.

If an error occurs in the `handleMessage()` method of the interceptor, or any subsequent interceptor in the interceptor chain, the `handleFault()` method is called. The `handleFault()` method is useful for cleaning up after an interceptor in the event of an error. It can also be used to alter the fault message.

Processing messages

Overview

In normal message processing, an interceptor's `handleMessage()` method is invoked. It receives that message data as a `Message` object. Along with the actual contents of the message, the `Message` object may contain a number of properties related to the message or the message processing state. The exact contents of the `Message` object depends on the interceptors preceding the current interceptor in the chain.

Getting the message contents

The `Message` interface provides two methods that can be used in extracting the message contents:

- ```
  public <T> T getContent(java.lang.Class<T>  format);
  ```

  The `getContent()` method returns the content of the message in an object of the specified class. If the contents are not available as an instance of the specified class, null is returned. The list of available content types is determined by the interceptor's location on the interceptor chain and the direction of the interceptor chain.

- ```
  public Collection<Attachment> getAttachments();
  ```

  The `getAttachments()` method returns a Java `Collection` object containing any binary attachments associated with the message. The attachments are stored in `org.apache.cxf.message.Attachment` objects. `Attachment` objects provide methods for managing the binary data.

Important

Attachments are only available after the attachment processing interceptors have executed.

Determining the message's direction

The direction of a message can be determined by querying the message exchange. The message exchange stores the inbound message and the outbound message in separate properties.[1]

The message exchange associated with a message is retrieved using the message's `getExchange()` method. As shown in Example 4.1 on page 29, `getExchange()` does not take any parameters and returns the message exchange as a `org.apache.cxf.message.Exchange` object.

[1] It also stores inbound and outbound faults separately.

Example 4.1. Getting the message exchange

```
Exchange getExchange();
```

The `Exchange` object has four methods, shown in Example 4.2 on page 29, for getting the messages associated with an exchange. Each method will either return the message as a `org.apache.cxf.Message` object or it will return null if the message does not exist.

Example 4.2. Getting messages from a message exchange

```
Message getInMessage();

Message getInFaultMessage();

Message getOutMessage();

Message getOutFaultMessage();
```

Example 4.3 on page 29 shows code for determining if the current message is outbound. The method gets the message exchange and checks to see if the current message is the same as the exchange's outbound message. It also checks the current message against the exchanges outbound fault message to error messages on the outbound fault interceptor chain.

Example 4.3. Checking the direction of a message chain

```
public static boolean isOutbound()
{
   Exchange exchange = message.getExchange();
        return message != null
               && exchange != null
               && (message == exchange.getOutMessage()
                   || message == exchange.getOutFaultMessage());
}
```

Example

Example 4.4 on page 29 shows code for an interceptor that processes zip compressed messages. It checks the direction of the message and then performs the appropriate actions.

Example 4.4. Example message processing method

```
import java.io.IOException;
import java.io.InputStream;
import java.util.zip.GZIPInputStream;

import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;
```

```
public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
{

  ...

  public void handleMessage(Message message)
  {

        boolean isOutbound = false;
        isOutbound = message == message.getExchange().getOutMessage()
               || message == message.getExchange().getOutFaultMessage();

        if (!isOutbound)
        {
          try
          {
             InputStream is = message.getContent(InputStream.class);
             GZIPInputStream zipInput = new GZIPInputStream(is);
             message.setContent(InputStream.class, zipInput);
          }
          catch (IOException ioe)
          {
             ioe.printStackTrace();
           }
        }
         else
        {
         // zip the outbound message
        }
    }
  ...
}
```

Unwinding after an error

Overview

When an error occurs during the execution of an interceptor chain, the runtime stops traversing the interceptor chain and unwinds the chain by calling the `handleFault()` method of any interceptors in the chain that have already been executed.

The `handleFault()` method can be used to clean up any resources used by an interceptor during normal message processing. It can also be used to rollback any actions that should only stand if message processing completes successfully. In cases where the fault message will be passed on to an outbound fault processing interceptor chain, the `handleFault()` method can also be used to add information to the fault message.

Getting the message payload

The `handleFault()` method receives the same `Message` object as the `handleMessage()` method used in normal message processing. Getting the message contents from the `Message` object is described in "Getting the message contents" on page 28.

Example

Example 4.5 on page 31 shows code used to ensure that the original XML stream is placed back into the message when the interceptor chain is unwound.

Example 4.5. Handling an unwinding interceptor chain

```
@Override
public void handleFault(SoapMessage message)
{
  super.handleFault(message);
  XMLStreamWriter writer = (XMLStreamWriter)message.get(ORIGINAL_XML_WRITER);
  if (writer != null)
  {
    message.setContent(XMLStreamWriter.class, writer);
  }
}
```

Chapter 5. Configuring Endpoints to Use Interceptors

Interceptors are added to an endpoint when it is included in a message exchange. The endpoint's interceptor chains are constructed from a the interceptor chains of a number of components in the Fuse Services Framework runtime. Interceptors are specified in either the endpoint's configuration or the configuration of one of the runtime components. Interceptors can be added using either the configuration file or the interceptor API.

Deciding where to attach interceptors

Overview

There are a number of runtime objects that host interceptor chains. These include:

- the endpoint object
- the service object
- the proxy object
- the factory object used to create the endpoint or the proxy
- the binding
- the central `Bus` object

A developer can attach their own interceptors to any of these objects. The most common objects to attach interceptors are the bus and the individual endpoints. Choosing the correct object requires understanding how these runtime objects are combined to make an endpoint.

Endpoints and proxies

Attaching interceptors to either the endpoint or the proxy is the most fine grained way to place an interceptor. Any interceptors attached directly to an endpoint or a proxy only effect the specific endpoint or proxy. This is a good place to attach interceptors that are specific to a particular incarnation of a service. For example, if a developer wants to expose one instance of a service that converts units from metric to imperial they could attach the interceptors directly to one endpoint.

Factories

Using the Spring configuration to attach interceptors to the factories used to create an endpoint or a proxy has the same effect as attaching the interceptors directly to the endpoint or proxy. However, when interceptors are attached to a factory programmatically the interceptors attached to the factory are propagated to every endpoint or proxy created by the factory.

Bindings

Attaching interceptors to the binding allows the developer to specify a set of interceptors that are applied to all endpoints that use the binding. For example, if a developer wants to force all endpoints that use the raw XML binding to include a special ID element, they could attach the interceptor responsible for adding the element to the XML binding.

Buses

The most general place to attach interceptors is the bus. When interceptors are attached to the bus, the interceptors are propagated to all of the endpoints managed by that bus. Attaching interceptors to the bus is useful in applications that create multiple endpoints that share a similar set of interceptors.

Combining attachment points

Because an endpoint's final set of interceptor chains is an amalgamation of the interceptor chains contributed by the listed objects, several of the listed object can be combined in a single endpoint's configuration. For example, if an application spawned multiple endpoints that all required an interceptor that checked for a validation token, that interceptor would be attached to the application's bus. If one of those endpoints also required an interceptor that converted Euros into dollars, the conversion interceptor would be attached directly to the specific endpoint.

Adding interceptors using configuration

Overview

The easiest way to attach interceptors to an endpoint is using the configuration file. Each interceptor to be attached to an endpoint is configured using a standard Spring bean. The interceptor's bean can then be added to the proper interceptor chain using Fuse Services Framework configuration elements.

Each runtime component that has an associated interceptor chain is configurable using specialized Spring elements. Each of the component's elements have a standard set of children for specifying their interceptor chains. There is one child for each interceptor chain associated with the component. The children list the beans for the interceptors to be added to the chain.

Configuration elements

Table 5.1 on page 36 describes the four configuration elements for attaching interceptors to a runtime component.

Table 5.1. Interceptor chain configuration elements

Element	Description
`inInterceptors`	Contains a list of beans configuring interceptors to add to an endpoint's inbound interceptor chain.
`outInterceptors`	Contains a list of beans configuring interceptors to add to an endpoint's outbound interceptor chain.
`inFaultInterceptors`	Contains a list of beans configuring interceptors to add to an endpoint's inbound fault processing interceptor chain.
`outFaultInterceptors`	Contains a list of beans configuring interceptors to add to an endpoint's outbound fault processing interceptor chain.

All of the interceptor chain configuration elements take a `list` child element. The `list` element has one child for each of the interceptors being attached to the chain. Interceptors can be specified using either a `bean` element directly configuring the interceptor or a `ref` element that refers to a `bean` element that configures the interceptor.

Examples

Example 5.1 on page 37 shows configuration for attaching interceptors to a bus' inbound interceptor chain.

Example 5.1. Attaching interceptors 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"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="
       http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
      http://cxf.apache.org/transports/http/configuration http://cxf.apache.org/schemas/con
figuration/http-conf.xsd
       http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans.xsd">
  ...
  <bean id="GZIPStream" class="demo.stream.interceptor.StreamInterceptor"/>

  <cxf:bus>
    <cxf:inInterceptors>
      <list>
        <ref bean="GZIPStream"/>
      </list>
    </cxf:inInterceptors>
  </cxf:bus>
</beans>
```

Example 5.2 on page 37 shows configuration for attaching an interceptor to a JAX-WS service's outbound interceptor chain.

Example 5.2. Attaching interceptors to a JAX-WS service provider

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

  <jaxws:endpoint ...>
    <jaxws:outInterceptors>
      <list>
        <bean id="GZIPStream" class="demo.stream.interceptor.StreamInterceptor" />
      </list>
    </jaxws:outInterceptors>
  </jaxws:endpoint>
</beans>
```

More information

For more information about configuring endpoints using the Spring configuration see *Configuring Web Service Endpoints*.

Adding interceptors programmatically

Interceptors can be attached to endpoints programmatically using either one of two approaches:

- the `InterceptorProvider` API
- Java annotations

Using the `InterceptorProvider` API allows the developer to attach interceptors to any of the runtime components that have interceptor chains, but it requires working with the underlying Fuse Services Framework classes. The Java annotations can only be added to service interfaces or service implementations, but they allow developers to stay within the JAX-WS API or the JAX-RS API.

Using the interceptor provider API

Overview

Interceptors can be registered with any component that implements the `InterceptorProvider` interface shown in Example 5.3 on page 40.

Example 5.3. The interceptor provider interface

```
package org.apache.cxf.interceptor;

import java.util.List;

public interface InterceptorProvider
{
    List<Interceptor<? extends Message>> getInInterceptors();

    List<Interceptor<? extends Message>> getOutInterceptors();

    List<Interceptor<? extends Message>> getInFaultInterceptors();

    List<Interceptor<? extends Message>> getOutFaultInterceptors();
}
```

The four methods in the interface allow you to retrieve each of an endpoint's interceptor chains as a Java `List` object. Using the methods offered by the Java `List` object, developers can add and remove interceptors to any of the chains.

Procedure

To use the `InterceptorProvider` API to attach an interceptor to a runtime component's interceptor chain, you must:

1. Get access to the runtime component with the chain to which the interceptor is being attached.

   Developers must use Fuse Services Framework specific APIs to access the runtime components from standard Java application code. The runtime components are usually accessible by casting the JAX-WS or JAX-RS artifacts into the underlying Fuse Services Framework objects.

2. Create an instance of the interceptor.

3. Use the proper get method to retrieve the desired interceptor chain.

4. Use the `List` object's `add()` method to attach the interceptor to the interceptor chain.

Tip

This step is usually combined with retrieving the interceptor chain.

Attaching an interceptor to a consumer

Example 5.4 on page 41 shows code for attaching an interceptor to the inbound interceptor chain of a JAX-WS consumer.

Example 5.4. Attaching an interceptor to a consumer programmatically

```
package com.fusesource.demo;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import org.apache.cxf.endpoint.ClientProxy;
import org.apache.cxf.endpoint.ClientProxy;

public class Client
{
  public static void main(String args[])
  {
    QName serviceName = new QName("http://demo.eric.org", "stockQuoteReporter");
    Service s = Service.create(serviceName); ❶

    QName portName = new QName("http://demo.eric.org", "stockQuoteReporterPort");
   s.addPort(portName, "http://schemas.xmlsoap.org/soap/", "http://localhost:9000/EricStock
Quote"); ❷

    quoteReporter proxy = s.getPort(portName, quoteReporter.class); ❸

    Client cxfClient = ClientProxy.getClient(proxy); ❹

    ValidateInterceptor validInterceptor = new ValidateInterceptor(); ❺
    cxfClient.getInInterceptor().add(validInterceptor); ❻

    ...
  }
}
```

The code in Example 5.4 on page 41 does the following:

❶ Creates a JAX-WS `Service` object for the consumer.
❷ Adds a port to the `Service` object that provides the consumer's target address.
❸ Creates the proxy used to invoke methods on the service provider.
❹ Gets the Fuse Services Framework `Client` object associated with the proxy.
❺ Creates an instance of the interceptor.
❻ Attaches the interceptor to the inbound interceptor chain.

Attaching an interceptor to a service provider

Example 5.5 on page 42 shows code for attaching an interceptor to a service provider's outbound interceptor chain.

Example 5.5. Attaching an interceptor to a service provider programmatically

```
package com.fusesource.demo;
import java.util.*;

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

public class stockQuoteReporter implements quoteReporter
{
  ...
  public stockQuoteReporter()
  {
    ServerFactoryBean sfb = new ServerFactoryBean(); ❶
    Server server = sfb.create(); ❷
    EndpointImpl endpt = server.getEndpoint(); ❸

    AuthTokenInterceptor authInterceptor = new AuthTokenInterceptor(); ❹

    endpt.getOutInterceptor().add(authInterceptor); ❺
  }
}
```

The code in Example 5.5 on page 42 does the following:

❶ Creates a `ServerFactoryBean` object that will provide access to the underlying Fuse Services Framework objects.
❷ Gets the `Server` object that Fuse Services Framework uses to represent the endpoint.
❸ Gets the Fuse Services Framework `EndpointImpl` object for the service provider.
❹ Creates an instance of the interceptor.
❺ Attaches the interceptor to the endpoint;s outbound interceptor chain.

Attaching an interceptor to a bus

Example 5.6 on page 43 shows code for attaching an interceptor to a bus' inbound interceptor chain.

Example 5.6. Attaching an interceptor to a bus

```
import org.apache.cxf.BusFactory;
org.apache.cxf.Bus;

...

Bus bus = BusFactory.getDefaultBus(); ❶

WatchInterceptor watchInterceptor = new WatchInterceptor(); ❷

bus..getInInterceptor().add(watchInterceptor); ❸

...
```

The code in Example 5.6 on page 43 does the following:

❶ Gets the default bus for the runtime instance.
❷ Creates an instance of the interceptor.
❸ Attaches the interceptor to the inbound interceptor chain.

The `WatchInterceptor` will be attached to the inbound interceptor chain of all endpoints created by the runtime instance.

Using Java annotations

Overview

Fuse Services Framework provides four Java annotations that allow a developer to specify the interceptor chains used by an endpoint. Unlike the other means of attaching interceptors to endpoints, the annotations are attached to application-level artifacts. The artifact that is used determines the scope of the annotation's effect.

Where to place the annotations

The annotations can be placed on the following artifacts:

- the service endpoint interface(SEI) defining the endpoint

  If the annotations are placed on an SEI, all of the service providers that implement the interface and all of the consumers that use the SEI to create proxies will be affected.

- a service implementation class

  If the annotations are placed on an implementation class, all of the service providers using the implementation class will be affected.

The annotations

The annotations are all in the `org.apache.cxf.interceptor` package and are described in Table 5.2 on page 44.

Table 5.2. Interceptor chain annotations

Annotation	Description
`InInterceptors`	Specifies the interceptors for the inbound interceptor chain.
`OutInterceptors`	Specifies the interceptors for the outbound interceptor chain.
`InFaultInterceptors`	Specifies the interceptors for the inbound fault interceptor chain.
`OutFaultInterceptors`	Specifies the interceptors for the outbound fault interceptor chain.

Listing the interceptors

The list of interceptors is specified as a list of fully qualified class names using the syntax shown in Example 5.7 on page 45.

Example 5.7. Syntax for listing interceptors in a chain annotation

```
interceptors={"interceptor1", "interceptor2", ..., "interceptorN"}
```

Example

Example 5.8 on page 45 shows annotations that attach two interceptors to the inbound interceptor chain of endpoints that use the logic provided by `SayHiImpl`.

Example 5.8. Attaching interceptors to a service implementation

```
import org.apache.cxf.interceptor.InInterceptors;

@InInterceptors(interceptors={"com.sayhi.interceptors.FirstLast", "com.sayhi.interceptors.LogName"})
public class SayHiImpl implements SayHi
{
  ...
}
```

Chapter 6. Manipulating Interceptor Chains on the Fly

Interceptors can reconfigure an endpoint's interceptor chain as part of its message processing logic. It can add new interceptors, remove interceptors, reorder interceptors, and even suspend the interceptor chain. Any on-the-fly manipulation is invocation-specific, so the original chain is used each time an endpoint is involved in a message exchange.

Overview

Interceptor chains only live as long as the message exchange that sparked their creation. Each message contains a reference to the interceptor chain responsible for processing it. Developers can use this reference to alter the message's interceptor chain. Because the chain is per-exchange, any changes made to a message's interceptor chain will not effect other message exchanges.

Chain life-cycle

Interceptor chains and the interceptors in the chain are instantiated on a per-invocation basis. When an endpoint is invoked to participate in a message exchange, the required interceptor chains are instantiated along with instances of its interceptors. When the message exchange that caused the creation of the interceptor chain is completed, the chain and its interceptor instances are destroyed.

This means that any changes you make to the interceptor chain or to the fields of an interceptor do not persist across message exchanges. So, if an interceptor places another interceptor in the active chain only the active chain is effected. Any future message exchanges will be created from a pristine state as determined by the endpoint's configuration. It also means that a developer cannot set flags in an interceptor that will alter future message processing.

Tip

If an interceptor needs to pass information along to future instances, it can set a property in the message context. The context does persist across message exchanges.

Getting the interceptor chain

The first step in changing a message's interceptor chain is getting the interceptor chain. This is done using the `Message.getInterceptorChain()` method shown in Example 6.1 on page 48. The interceptor chain is returned as a `org.apache.cxf.interceptor.InterceptorChain` object.

Example 6.1. Method for getting an interceptor chain

```
InterceptorChain getInterceptorChain();
```

Adding interceptors

The `InterceptorChain` object has two methods, shown in Example 6.2 on page 48, for adding interceptors to an interceptor chain. One allows you to add a single interceptor and the other allows you to add multiple interceptors.

Example 6.2. Methods for adding interceptors to an interceptor chain

```
void add(Interceptor<? extends Message> i);

void add(Collection<Interceptor<? extends Message>> i);
```

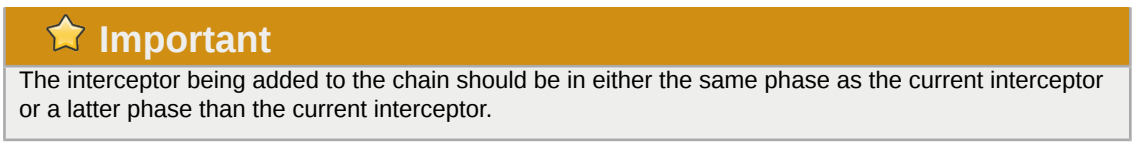
Example 6.3 on page 48 shows code for adding a single interceptor to a message's interceptor chain.

Example 6.3. Adding an interceptor to an interceptor chain on-the-fly

```
void handleMessage(Message message)
{
  ...
  AddledIntereptor addled = new AddledIntereptor(); ❶
  InterceptorChain chain = message.getInterceptorChain(); ❷
  chain.add(addled); ❸
  ...
}
```

The code in Example 6.3 on page 48 does the following:

❶ Instantiates a copy of the interceptor to be added to the chain.



❷ Gets the interceptor chain for the current message.
❸ Adds the new interceptor to the chain.

Removing interceptors

The `InterceptorChain` object has one method, shown in Example 6.4 on page 49, for removing an interceptor from an interceptor chain.

Example 6.4. Methods for removing interceptors from an interceptor chain

```
void remove(Interceptor<? extends Message> i);
```

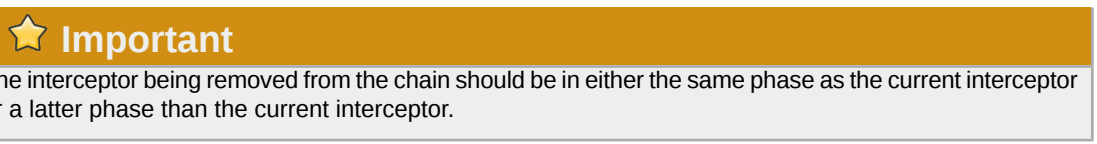
Example 6.5 on page 49 shows code for removing an interceptor from a message's interceptor chain.

Example 6.5. Removing an interceptor from an interceptor chain on-the-fly

```
void handleMessage(Message message)
{
  ...
  SackedIntereptor sacked = new SackedIntereptor(); ❶
  InterceptorChain chain = message.getInterceptorChain(); ❷
  chain.remove(sacked); ❸
  ...
}
```

The code in Example 6.5 on page 49 does the following:

❶ Instantiates a copy of the interceptor to be removed from the chain.



❷ Gets the interceptor chain for the current message.
❸ Removes the interceptor from the chain.

Appendix A. Fuse Services Framework Message Processing Phases

Inbound phases

Table A.1 on page 51 lists the phases available in inbound interceptor chains.

Table A.1. Inbound message processing phases

Phase	Description
RECEIVE	Performs transport specific processing, such as determining MIME boundaries for binary attachments.
PRE_STREAM	Processes the raw data stream received by the transport.
USER_STREAM	
POST_STREAM	
READ	Determines if a request is a SOAP or XML message and builds adds the proper interceptors. SOAP message headers are also processed in this phase.
PRE_PROTOCOL	Performs protocol level processing. This includes processing of WS-* headers and processing of the SOAP message properties.
USER_PROTOCOL	
POST_PROTOCOL	
UNMARSHAL	Unmarshals the message data into the objects used by the application level code.
PRE_LOGICAL	Processes the unmarshalled message data.
USER_LOGICAL	
POST_LOGICAL	
PRE_INVOKE	
INVOKE	Passes the message to the application code. On the server side, the service implementation is invoked in this phase. On the client side, the response is handed back to the application.
POST_INVOKE	Invokes the outbound interceptor chain.

Outbound phases

Table A.2 on page 52 lists the phases available in inbound interceptor chains.

Table A.2. Inbound message processing phases

Phase	Description
SETUP	Performs any set up that is required by later phases in the chain.
PRE_LOGICAL	Performs processing on the unmarshalled data passed from the application level.
USER_LOGICAL	
POST_LOGICAL	
PREPARE_SEND	Opens the connection for writing the message on the wire.
PRE_STREAM	Performs processing required to prepare the message for entry into a data stream.
PRE_PROTOCOL	Begins processing protocol specific information.
WRITE	Writes the protocol message.
PRE_MARSHAL	Marshals the message.
MARSHAL	
POST_MARSHAL	
USER_PROTOCOL	Process the protocol message.
POST_PROTOCOL	
USER_STREAM	Process the byte-level message.
POST_STREAM	
SEND	Sends the message and closes the transport stream.

Important

Outbound interceptor chains have a mirror set of ending phases whose names are appended with `_ENDING`. The ending phases are used interceptors that require some terminal action to occur before data is written on the wire.

Appendix B. Fuse Services Framework Provided Interceptors

Core Fuse Services Framework Interceptors

Inbound

Table B.1 on page 54 lists the core inbound interceptors that are added to all Fuse Services Framework endpoints.

Table B.1. Core inbound interceptors

Class	Phase	Description
`ServiceInvokerInterceptor`	`INVOKE`	Invokes the proper method on the service.

Outbound

The Fuse Services Framework does not add any core interceptors to the outbound interceptor chain by default. The contents of an endpoint's outbound interceptor chain depend on the features in use.

Front-Ends

JAX-WS

Table B.2 on page 55 lists the interceptors added to a JAX-WS endpoint's inbound message chain.

Table B.2. Inbound JAX-WS interceptors

Class	Phase	Description
`HolderInInterceptor`	`PRE_INVOKE`	Creates holder objects for any out or in/out parameters in the message.
`WrapperClassInInterceptor`	`POST_LOGICAL`	Unwraps the parts of a wrapped doc/literal message into the appropriate array of objects.
`LogicalHandlerInInterceptor`	`PRE_PROTOCOL`	Passes message processing to the JAX-WS logical handlers used by the endpoint. When the JAX-WS handlers complete, the message is passed along to the next interceptor on the inbound chain.
`SOAPHandlerInterceptor`	`PRE_PROTOCOL`	Passes message processing to the JAX-WS SOAP handlers used by the endpoint. When the SOAP handlers finish with the message, the message is passed along to the next interceptor in the chain.

Table B.3 on page 55 lists the interceptors added to a JAX-WS endpoint's outbound message chain.

Table B.3. Outbound JAX-WS interceptors

Class	Phase	Description
`HolderOutInterceptor`	`PRE_LOGICAL`	Removes the values of any out and in/out parameters from their holder objects and adds the values to the message's parameter list.
`WebFaultOutInterceptor`	`PRE_PROTOCOL`	Processes outbound fault messages.
`WrapperClassOutInterceptor`	`PRE_LOGICAL`	Makes sure that wrapped doc/literal messages and rpc/literal messages

Class	Phase	Description
		are properly wrapped before being added to the message.
`LogicalHandlerOutInterceptor`	`PRE_MARSHAL`	Passes message processing to the JAX-WS logical handlers used by the endpoint. When the JAX-WS handlers complete, the message is passed along to the next interceptor on the outbound chain.
`SOAPHandlerInterceptor`	`PRE_PROTOCOL`	Passes message processing to the JAX-WS SOAP handlers used by the endpoint. When the SOAP handlers finish processing the message, it is passed along to the next interceptor in the chain.
`MessageSenderInterceptor`	`PREPARE_SEND`	Calls back to the Destination object to have it setup the output streams, headers, etc. to prepare the outgoing transport.

JAX-RS

Table B.4 on page 56 lists the interceptors added to a JAX-RS endpoint's inbound message chain.

Table B.4. Inbound JAX-RS interceptors

Class	Phase	Description
`JAXRSInInterceptor`	`PRE_STREAM`	Selects the root resource class, invokes any configured JAX-RS request filters, and determines the method to invoke on the root resource.

Important

The inbound chain for a JAX-RS endpoint skips straight to the `ServiceInvokerInInterceptor` interceptor. No other interceptors will be invoked after the `JAXRSInInterceptor`.

Table B.5 on page 57 lists the interceptors added to a JAX-RS endpoint's outbound message chain.

Table B.5. Outbound JAX-RS interceptors

Class	Phase	Description
`JAXRSOutInterceptor`	`MARSHAL`	Marshals the response into the proper format for transmission.

Message bindings

SOAP

Table B.6 on page 58 lists the interceptors added to a endpoint's inbound message chain when using the SOAP Binding.

Table B.6. Inbound SOAP interceptors

Class	Phase	Description
`CheckFaultInterceptor`	`POST_PROTOCOL`	Checks if the message is a fault message. If the message is a fault message, normal processing is aborted and fault processing is started.
`MustUnderstandInterceptor`	`PRE_PROTOCOL`	Processes the must understand headers.
`RPCInInterceptor`	`UNMARSHAL`	Unmarshals rpc/literal messages. If the message is bare, the message is passed to a `BareInInterceptor` object to deserialize the message parts.
`ReadsHeadersInterceptor`	`READ`	Parses the SOAP headers and stores them in the message object.
`SoapActionInInterceptor`	`READ`	Parses the SOAP action header and attempts to find a unique operation for the action.
`SoapHeaderInterceptor`	`UNMARSHAL`	Binds the SOAP headers that map to operation parameters to the appropriate objects.
`AttachmentInInterceptor`	`RECEIVE`	Parses 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 `Attachment` objects.
`DocLiteralInInterceptor`	`UNMARSHAL`	Examines the first element in the SOAP body to determine the appropriate operation and calls the data binding to read in the data.

Class	Phase	Description
StaxInInterceptor	POST_STREAM	Creates an XMLStreamReader object from the message.
URIMappingInterceptor	UNMARSHAL	Handles the processing of HTTP GET methods.
SwAInInterceptor	PRE_INVOKE	Creates the required MIME handlers for binary SOAP attachments and adds the data to the parameter list.

Table B.7 on page 59 lists the interceptors added to a endpoint's outbound message chain when using the SOAP Binding.

Table B.7. Outbound SOAP interceptors

Class	Phase	Description
RPCOutInterceptor	MARSHAL	Marshals rpc style messages for transmission.
SoapHeaderOutFilterInterceptor	PRE_LOGICAL	Removes all SOAP headers that are marked as inbound only.
SoapPreProtocolOutInterceptor	POST_LOGICAL	Sets up the SOAP version and the SOAP action header.
AttachmentOutInterceptor	PRE_STREAM	Sets up the attachment marshalers and the mime stuff required to process any attachments that might be in the message.
BareOutInterceptor	MARSHAL	Writes the message parts.
StaxOutInterceptor	PRE_STREAM	Creates an XMLStreamWriter object from the message.
WrappedOutInterceptor	MARSHAL	Wraps the outbound message parameters.
SoapOutInterceptor	WRITE	Writes the soap:envelope element and the elements for the header blocks in the message. Also writes an empty soap:body element for the remaining interceptors to populate.
SwAOutInterceptor	PRE_LOGICAL	Removes any binary data that will be packaged as a SOAP attachment and stores it for later processing.

XML

Table B.8 on page 60 lists the interceptors added to a endpoint's inbound message chain when using the XML Binding.

Table B.8. Inbound XML interceptors

Class	Phase	Description
`AttachmentInInterceptor`	`RECEIVE`	Parses the mime headers for mime boundaries, finds the *root* part and resets the input stream to it, and then stores the other parts in a collection of `Attachment` objects.
`DocLiteralInInterceptor`	`UNMARSHAL`	Examines the first element in the message body to determine the appropriate operation and then calls the data binding to read in the data.
`StaxInInterceptor`	`POST_STREAM`	Creates an `XMLStreamReader` object from the message.
`URIMappingInterceptor`	`UNMARSHAL`	Handles the processing of HTTP GET methods.
`XMLMessageInInterceptor`	`UNMARSHAL`	Unmarshals the XML message.

Table B.9 on page 60 lists the interceptors added to a endpoint's outbound message chain when using the XML Binding.

Table B.9. Outbound XML interceptors

Class	Phase	Description
`StaxOutInterceptor`	`PRE_STREAM`	Creates an `XMLStreamWriter` objects from the message.
`WrappedOutInterceptor`	`MARSHAL`	Wraps the outbound message parameters.
`XMLMessageOutInterceptor`	`MARSHAL`	Marshals the message for transmission.

CORBA

Table B.10 on page 61 lists the interceptors added to a endpoint's inbound message chain when using the CORBA Binding.

Table B.10. Inbound CORBA interceptors

Class	Phase	Description
`CorbaStreamInInterceptor`	`PRE_STREAM`	Deserializes the CORBA message.
`BareInInterceptor`	`UNMARSHAL`	Deserializes the message parts.

Table B.11 on page 61 lists the interceptors added to a endpoint's outbound message chain when using the CORBA Binding.

Table B.11. Outbound CORBA interceptors

Class	Phase	Description
`CorbaStreamOutInterceptor`	`PRE_STREAM`	Serializes the message.
`BareOutInterceptor`	`MARSHAL`	Writes the message parts.
`CorbaStreamOutEndingInterceptor`	`USER_STREAM`	Creates a streamable object for the message and stores it in the message context.

Other features

Logging

Table B.12 on page 62 lists the interceptors added to a endpoint's inbound message chain to support logging.

Table B.12. Inbound logging interceptors

Class	Phase	Description
`LoggingInInterceptor`	`RECEIVE`	Writes the raw message data to the logging system.

Table B.13 on page 62 lists the interceptors added to a endpoint's outbound message chain to support logging.

Table B.13. Outbound logging interceptors

Class	Phase	Description
`LoggingOutInterceptor`	`PRE_STREAM`	Writes the outbound message to the logging system.

For more information about logging see "Fuse Services Framework Logging" in *Configuring Web Service Endpoints*.

WS-Addressing

Table B.14 on page 62 lists the interceptors added to a endpoint's inbound message chain when using WS-Addressing.

Table B.14. Inbound WS-Addressing interceptors

Class	Phase	Description
`MAPCodec`	`PRE_PROTOCOL`	Decodes the message addressing properties.

Table B.15 on page 62 lists the interceptors added to a endpoint's outbound message chain when using WS-Addressing.

Table B.15. Outbound WS-Addressing interceptors

Class	Phase	Description
`MAPAggregator`	`PRE_LOGICAL`	Aggregates the message addressing properties for a message.

Class	Phase	Description
`MAPCodec`	`PRE_PROTOCOL`	Encodes the message addressing properties.

For more information about WS-Addressing see "Deploying WS-Addressing" in *Configuring Web Service Endpoints*.

WS-RM

Important

WS-RM relies on WS-Addressing so all of the WS-Addressing interceptors will also be added to the interceptor chains.

Table B.16 on page 63 lists the interceptors added to a endpoint's inbound message chain when using WS-RM.

Table B.16. Inbound WS-RM interceptors

Class	Phase	Description
`RMInInterceptor`	`PRE_LOGICAL`	Handles the aggregation of message parts and acknowledgement messages.
`RMSoapInterceptor`	`PRE_PROTOCOL`	Encodes and decodes the WS-RM properties from messages.

Table B.17 on page 63 lists the interceptors added to a endpoint's outbound message chain when using WS-RM.

Table B.17. Outbound WS-RM interceptors

Class	Phase	Description
`RMOutInterceptor`	`PRE_LOGICAL`	Handles the chunking of messages and the transmission of the chunks. Also handles the processing of acknowledgements and resend requests.
`RMSoapInterceptor`	`PRE_PROTOCOL`	Encodes and decodes the WS-RM properties from messages.

For more information about WS-RM see "Enabling Reliable Messaging" in *Configuring Web Service Endpoints*.

Appendix C. Interceptor Providers

Overview

Interceptor providers are objects in the Fuse Services Framework runtime that have interceptor chains attached to them. They all implement the `org.apache.cxf.interceptor.InterceptorProvider` interface. Developers can attach their own interceptors to any interceptor provider.

List of providers

The following objects are interceptor providers:

- `AddressingPolicyInterceptorProvider`
- `ClientFactoryBean`
- `ClientImpl`
- `ClientProxyFactoryBean`
- `CorbaBinding`
- `CXFBusImpl`
- `org.apache.cxf.jaxws.EndpointImpl`
- `org.apache.cxf.endpoint.EndpointImpl`
- `ExtensionManagerBus`
- `JAXRSClientFactoryBean`
- `JAXRSServerFactoryBean`
- `JAXRSServiceImpl`
- `JaxWsClientEndpointImpl`
- `JaxWsClientFactoryBean`
- `JaxWsEndpointImpl`
- `JaxWsProxyFactoryBean`
- `JaxWsServerFactoryBean`

- JaxwsServiceBuilder
- MTOMPolicyInterceptorProvider
- NoOpPolicyInterceptorProvider
- ObjectBinding
- RMPolicyInterceptorProvider
- ServerFactoryBean
- ServiceImpl
- SimpleServiceBuilder
- SoapBinding
- WrappedEndpoint
- WrappedService
- XMLBinding

Index

Symbols

A

C

E

H

I

M

O

P