Red Hat JBoss Fuse 6.3

Smooks Development Guide

Manipulate messages using Smooks

Last Updated: 2018-07-16

Red Hat JBoss Fuse 6.3 Smooks Development Guide

Manipulate messages using Smooks

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

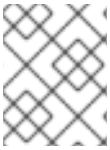
(Deprecated since JBoss Fuse 6.3) Use this guide to help you develop integrated applications with SwitchYard.

Table of Contents

CHAPTER 1. BASICS

1.1. SMOOKS

Smooks is a fragment-based data transformation and analysis tool. It is a general purpose processing tool capable of interpreting fragments of a message. It uses visitor logic to accomplish this. It allows you to implement your transformation logic in XSLT or Java and provides a management framework through which you can centrally manage the transformation logic for your message-set.



NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

1.2. VISITOR LOGIC IN SMOOKS

Smooks uses *visitor logic*. A "visitor" is Java code that performs a specific action on a specific fragment of a message. This enables Smooks to perform actions on message fragments.

1.3. MESSAGE FRAGMENT PROCESSING

Smooks supports these types of message fragment processing:

- **`Templating:`** Transform message fragments with XSLT or FreeMarker
- **`Java Binding:`** Bind message fragment data into Java objects
- **`Splitting:`** Split messages fragments and rout the split fragments over multiple transports and destinations
- **`Enrichment:`** "Enrich" message fragments with data from databases
- **`Persistence:`** Persist message fragment data to databases
- **`Validation:`** Perform basic or complex validation on message fragment data

1.4. BASIC PROCESSING MODEL

The following is a list of different transformations you can perform with Smooks:

- XML to XML
- XML to Java
- Java to XML
- Java to Java
- EDI to XML
- EDI to Java
- Java to EDI

- CSV to XML

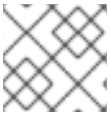
1.5. SUPPORTED MODELS

Simple API for XML (SAX)

The SAX event model is based on the hierarchical SAX events you can generate from an XML source. These include the **`startElement`** and **`endElement`**. Apply it to other structured and hierarchical data sources like **`EDI`**, **`CSV`** and Java files.

Document Object Model (DOM)

Use this object model to map the message source and its final result.



NOTE

The most important events have **`visitBefore`** and **`visitAfter`** in their titles.

1.6. FREEMARKER

FreeMarker is a template engine. You can use it to create and use a **`NodeModel`** as the domain model for a template operation. Smooks adds the ability to perform fragment-based template transformations to this functionality, as well as the power to apply the model to huge messages.

1.7. EXAMPLE OF USING SAX

Prerequisites

- Requires an implemented *SAXVisitor* interface. (Choose an interface that corresponds to the events of the process.)
- This example uses the **`ExecutionContext`** name. It is a *public interface* which extends the **`BoundAttributeStore`** class.

Procedure 1.1. Task

1. Create a new Smooks configuration. This will be used to apply the visitor logic at the <xxx> element's **`visitBefore`** and **`visitAfter`** events.
2. Apply the logic at the **`visitBefore`** and **`visitAfter`** events in a specific element of the overall event stream. The visitor logic is applied to the events in the <xxx> element.
3. Use Smooks with **FreeMarker** to perform an XML-to-XML transformation on a huge message.
4. Insert the following source format:

```
<order id='332'>
    <header>
        <customer number="123">Joe</customer>
    </header>
    <order-items>
        <order-item id='1'>
            <product>1</product>
```

```
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>

        <!-- etc etc -->

    </order-items>
</order>
```

5. Insert this target format:

```
<salesorder>
    <details>
        <orderid>332</orderid>
        <customer>
            <id>123</id>
            <name>Joe</name>
        </customer>
    <details>
    <itemList>
        <item>
            <id>1</id>
            <productId>1</productId>
            <quantity>2</quantity>
            <price>8.80</price>
        <item>

        <!-- etc etc -->

    </itemList>
</salesorder>
```

6. Use this Smooks configuration:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Filter the message using the SAX Filter (i.e. not DOM, so no
    intermediate DOM for the "complete" message - there are "mini"
DOMs
    for the NodeModels below)....
    -->
    <params>
        <param name="stream.filter.type">SAX</param>
        <param name="default.serialization.on">false</param>
    </params>

    <!--
    Create 2 NodeModels.  One high level model for the "order"
    (header etc) and then one per "order-item".
```

```
    These models are used in the FreeMarker templating resources
    defined below.  You need to make sure you set the selector such
    that the total memory footprint is as low as possible.  In this
    example, the "order" model will contain everything accept the
    <order-item> data (the main bulk of data in the message).  The
    "order-item" model only contains the current <order-item> data
    (i.e. there's max 1 order-item in memory at any one time).
    -->
    <resource-config selector="order,order-item">
        <resource>org.milyn.delivery.DomModelCreator</resource>
    </resource-config>

    <!--
    Apply the first part of the template when we reach the start
    of the <order-items> element.  Apply the second part when we
    reach the end.

    Note the <?TEMPLATE-SPLIT-PI?> Processing Instruction in the
    template.  This tells Smooks where to split the template,
    resulting in the order-items being inserted at this point.
    -->
    <ftl:freemarker applyOnElement="order-items">
        <ftl:template><!--<salesorder>
    <details>
        <orderid>${order.@id}</orderid>
        <customer>
            <id>${order.header.customer.@number}</id>
            <name>${order.header.customer}</name>
        </customer>
    </details>
    <itemList>
        <?TEMPLATE-SPLIT-PI?>
    </itemList>
</salesorder>--></ftl:template>
    </ftl:freemarker>

    <!--
    Output the <order-items> elements.  This will appear in the
    output message where the <?TEMPLATE-SPLIT-PI?> token appears in
the
    order-items template.
    -->
    <ftl:freemarker applyOnElement="order-item">
        <ftl:template><!--        <item>
            <id>${.vars["order-item"].@id}</id>
            <productId>${.vars["order-item"].product}</productId>
            <quantity>${.vars["order-item"].quantity}</quantity>
            <price>${.vars["order-item"].price}</price>
        </item>
        --></ftl:template>
    </ftl:freemarker>

</smooks-resource-list>
```

7. Use this code to execute:

```
Smooks smooks = new Smooks("smooks-config.xml");
try {
    smooks.filterSource(new StreamSource(new FileInputStream("input-
message.xml")), new StreamResult(System.out));
} finally {
    smooks.close();
}
```

8. An XML-to-XML transformation occurs as a result.

1.8. CARTRIDGES

A *cartridge* is a *Java archive* (JAR) file that contains reusable *content handlers*. In most cases, you will not need to write large quantities of Java code for Smooks because some modules of functionality are included as cartridges. You can create new cartridges of your own to extend the smooks-core's basic functionality. Each cartridge provides ready-to-use support for either a transformation process or a specific form of XML analysis.

1.9. SUPPLIED CARTRIDGES

These are the cartridges supplied with Smooks:

- Calc:"milyn-smooks-calc"
- CSV: "milyn-smooks-csv"
- Fixed length reader: "milyn-smooks-fixed-length"
- EDI: "milyn-smooks-edi"
- Javabean: "milyn-smooks-javabean"
- JSON: "milyn-smooks-json"
- Routing: "milyn-smooks-routing"
- Templating: "milyn-smooks-templating"
- CSS: "milyn-smooks-css"
- Servlet: "milyn-smooks-servlet"
- Persistence: "milyn-smooks-persistence"
- Validation: "milyn-smooks-validation"

1.10. SELECTORS

Smooks resource selectors tell Smooks which messages fragments to apply visitor logic. They also serve as simple look-up values for non-visitor logic. When a resource is a visitor implementation (like **<jb:bean>** or **<ftl:freemarker>**), Smooks treats the resource selector as an XPath selector. Resources include the **Java Binding Resource** and **FreeMarker Template Resource**.

1.11. USING SELECTORS

The following points apply when using the selectors:

- Configurations are both "strongly typed" and domain-specific for legibility.
- Configurations are XSD-based. This provides you with auto-completion support when using an *integrated development environment*.
- The actual handler doesn't need to be defined for the given resource type (such as the **`BeanPopulator`** class for Java bindings).

1.12. DECLARING NAMESPACES

Procedure 1.2. Task

- Configure namespace prefix-to-URI mappings through the core configuration namespace and modify the following XML code to include the namespaces you wish to use:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"
    xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-
1.3.xsd">

    <core:namespaces>
        <core:namespace prefix="a" uri="http://a"/>
        <core:namespace prefix="b" uri="http://b"/>
        <core:namespace prefix="c" uri="http://c"/>
        <core:namespace prefix="d" uri="http://d"/>
    </core:namespaces>

    <resource-config selector="c:item[@c:code =
'8655']/d:units[text() = 1]">
        <resource>com.acme.visitors.MyCustomVisitorImpl</resource>
    </resource-config>

</smooks-resource-list>
```

1.13. FILTERING PROCESS SELECTION

This is how Smooks selects a *filtering process*:

- The DOM processing model is selected automatically if only the DOM visitor interface is applied (**`DOMElementVisitor`** and **`SerializationUnit`**).
- If all visitor resources use only the SAX visitor interface (**`SAXElementVisitor`**), the SAX processing model is selected automatically.
- If the visitor resources use both the DOM and SAX interfaces, the DOM processing model is selected by default unless you specify SAX in the Smooks resource configuration file. This is done using **`<core:filterSettings type="SAX" />`**.

Visitor resources do not include *non-element visitor resources* such as **`readers`**.

1.14. EXAMPLE OF SETTING THE FILTER TYPE TO SAX IN SMOOKS

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

    <core:filterSettings type="SAX" />

</smooks-resource-list>
```

1.15. DOMMODELCREATOR

The DomModelCreator is a class that you can use in Smooks to create models for message fragments.

1.16. MIXING THE DOM AND SAX MODELS

- Use the DOM (Document Object Model) for *node traversal* (sending information between nodes) and pre-existing scripting/template engines.
- Use the **`DomModelCreator`** visitor class to mix SAX and DOM models. When used with SAX filtering, this visitor will construct a DOM fragment from the visited element. It allows you to use DOM utilities within a streaming environment.
- When more than one model is nested, the outer models will never contain data from the inner models (that is, the same fragment will never co-exist inside two models):

```
<order id="332">
    <header>
        <customer number="123">Joe</customer>
    </header>
    <order-items>
        <order-item id='1'>
            <product>1</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
        <order-item id='2'>
            <product>2</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
        <order-item id='3'>
            <product>3</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>
    </order-items>
</order>
```

1.17. CONFIGURING THE DOMMODELCREATOR

1. Configure the **DomModelCreator** from within Smooks to create models for the order and order-item message fragments. See the following example:

```
<resource-config selector="order,order-item">
    <resource>org.milyn.delivery.DomModelCreator</resource>
</resource-config>
```

2. Configure the in-memory model for the **`order`** as shown:

```
<order id='332'>
     <header>
         <customer number="123">Joe</customer>
     </header>
     <order-items />
</order>
```

NOTE

Each new model overwrites the previous one so there will never be more than one **`order-item`** model in memory at once.

1.18. FURTHER INFORMATION ABOUT THE DOMMODELCREATOR

- Groovy scripting: http://www.smooks.org/mediawiki/index.php?title=V1.3:groovy
- FreeMarker templates: http://www.smooks.org/mediawiki/index.php?title=V1.3:xml-to-xml

1.19. THE BEAN CONTEXT

The *bean context* contains objects for Smooks to access when filtering occurs. One bean context is created per execution context (using the **`Smooks.filterSource`** operation). Every bean the cartridge creates is filed according to its beanId.

1.20. CONFIGURING BEAN CONTEXTS

1. To have the contents of the bean context returned at the end of a **`Smooks.filterSource`** process, supply a **`org.milyn.delivery.java.JavaResult`** object in the call to the **`Smooks.filterSource`** method.

Example 1.1.

```
// Get the data to filter
StreamSource source = new
StreamSource(getClass().getResourceAsStream("data.xml"));

// Create a Smooks instance (cachable)
Smooks smooks = new Smooks("smooks-config.xml");

// Create the JavaResult, which will contain the filter result
after filtering
JavaResult result = new JavaResult();

// Filter the data from the source, putting the result into the
JavaResult
smooks.filterSource(source, result);
```

```
// Getting the Order bean which was created by the Javabean
cartridge
Order order = (Order)result.getBean("order");
```

2. To access the bean contexts at start-up, specify this in the **BeanContext** object. You can retrieve it from the **ExecutionContext** via the **getBeanContext()** method.

Example 1.2.

```
// Create a bean to pass on to the Smooks execution context
HashMap<String,Object> transformConfig = new
HashMap<String,Object>();
transformConfig.put("Version", new Integer(1));

// Get the data to filter
StreamSource source = new
StreamSource(getClass().getResourceAsStream("data.xml"));

// Create a Smooks instance (cachable)
Smooks smooks = new Smooks("smooks-config.xml");

// Create the JavaResult, which will contain the filter result
after filtering
JavaResult result = new JavaResult();

// Add bean to Smooks execution context
executionContext.getBeanContext().addBean("transformConfig",
transformConfig);

// Filter the data from the source, putting the result into the
JavaResult
smooks.filterSource(source, result);

// Getting the Order bean which was created by the JavaBean
cartridge
Order order = (Order)result.getBean("order");
```

3. When adding or retrieving objects from the **BeanContext** make sure you first retrieve a **beanId** object from the **beanIdStore**. The **beanId** object is a special key that ensures higher performance than string keys, although string keys are also supported.

4. You must retrieve the **beanIdStore** from the **ApplicationContext** using the **getbeanIdStore()** method.

5. To create a **beanId** object, call the **register("beanId name")** method. If you know that the beanId is already registered, then you can retrieve it by calling the **getbeanId("beanId name")** method.

6. **beanId** objects are **ApplicationContext**-scoped objects. Register them in your custom visitor implementation's initialization method and then put them in the visitor object as properties. You can then use them in the **visitBefore** and **visitAfter** methods. The **beanId** objects

and the **`beanIdStore`** are thread-safe.

1.21. PRE-INSTALLED BEANS

The following Beans come pre-installed:

- **`PUUID`**: UniqueId bean. This bean provides unique identifiers for the filtering **`ExecutionContext`**.
- **`PTIME`**: Time bean. This bean provides time-based data for the filtering**`ExecutionContext`**.

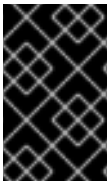
These examples show you how to use these beans in a **FreeMarker** template:

- Unique ID of the ExecutionContext (message being filtered): **`$PUUID.execContext`**
- Random Unique ID: **`$PUUID.random`**
- Message Filtering start time (in milliseconds): **`$PTIME.startMillis`**
- Message Filtering start time (in nanoseconds): **`$PTIME.startNanos`**
- Message Filtering start time (Date): **`$PTIME.startDate`**
- Time now (in milliseconds): **`$PTIME.nowMillis`**
- Time now (in nanoseconds): **`$PTIME.nowNanos`**
- Time now (Date): **`$PTIME.nowDate`**

1.22. MULTIPLE OUTPUTS/RESULTS

Smooks produces output in these ways:

- Through in-result instances. These are returned in the result instances passed to the **`Smooks.filterSource`** method.
- During the filtering process. This is achieved through output generated and sent to external endpoints (such as files, JMS destinations and databases) during the filtering process. Message fragment events trigger automatic routing to external endpoints.



IMPORTANT

Smooks can generate output in the above ways in a single filtering pass of a message stream. It does not need to filter a message stream multiple times to generate multiple outputs.

1.23. CREATING "IN-RESULT" INSTANCES

- Supply Smooks with multiple result instances as seen in the API:

```
public void filterSource(Source source, Result... results) throws
SmooksException
```

NOTE

Smooks does not support capturing result data from multiple result instances of the same type. For example, you can specify multiple StreamResult instances in the **`Smooks.filterSource`** method call, but Smooks will only output to one of these StreamResult instances (the first one).

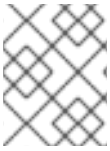
1.24. SUPPORTED RESULT TYPES

Smooks can work with standard **`JDK StreamResult`** and **`DOMResult`** result types, as well as these specialist ones:

- **`JavaResult`**: use this result type to capture the contents of the Smooks Java Bean context.
- **`ValidationResult`**: use this result type to capture outputs.
- Simple Result type: use this when writing tests. This is a **`StreamResult`** extension wrapping a **`StringWriter`**.

1.25. EVENT STREAM RESULTS

When Smooks processes a message, it produces a stream of events. If a **`StreamResult`** or **`DOMResult`** is supplied in the **`Smooks.filterSource`** call, Smooks will, by default, serialize the event stream (produced by the Source) to the supplied result as XML. (You can apply visitor logic to the event stream before serialization.)



NOTE

This is the mechanism used to perform a standard 1-input/1-xml-output character based transformation.

1.26. DURING THE FILTERING PROCESS

Smooks generates different types of output during the **`Smooks.filterSource`** process. (This occurs during the message event stream, before the end of the message is reached.) An example of this is when it is used to split and route message fragments to different types of endpoints for execution by other processes.

Smooks does not "batch up" the message data and produce all of the outputs after filtering the complete message. This is because performance would be impacted and also because it allows you to utilize the message event stream to trigger the fragment transformation and routing operations. Large messages are sent by streaming the process.

1.27. CHECKING THE SMOOKS EXECUTION PROCESS

1. To obtain an execution report from Smooks you must configure the **`ExecutionContext`** class to produce one. (Smooks will publish events as it processes messages.) The following sample code shows you how to configure Smooks to generate a HTML report:

```
Smooks smooks = new Smooks("/smooks/smooks-transform-x.xml");
ExecutionContext execContext = smooks.createExecutionContext();

execContext.setEventListener(new HtmlReportGenerator("/tmp/smooks-
```

```
report.html"));
smooks.filterSource(execContext, new StreamSource(inputStream), new
StreamResult(outputStream));
```

2. Use the **`HtmlReportGenerator`** feature to assist you when debugging.

NOTE

You can see a sample report on this web page:
http://www.milyn.org/docs/smooks-report/report.html

NOTE

Alternatively, you can create a custom **`ExecutionEventListener`** implementation.

1.28. TERMINATING THE FILTERING PROCESS

1. To terminate the Smooks filtering process before the end of the message is reached, add the <**`core:terminate`**> configuration to the Smooks settings. (This works for SAX and is not needed for DOM.)

   Here is an example configuration that terminates filtering at the end of the message's customer fragment:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"
   xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

    <!-- Visitors... -->
    <core:terminate onElement="customer" />

</smooks-resource-list>
```

2. To terminate at the beginning of a message (on the **`visitBefore`** event), use this code:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"
   xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">

   <!-- Visitors... -->

   <core:terminate onElement="customer" terminateBefore="true" />

</smooks-resource-list>
```

1.29. GLOBAL CONFIGURATION SETTINGS

Default Properties

Default Properties specify the default values for **<resource-config>** attributes. These properties are automatically applied to the **SmooksResourceConfiguration** class when the corresponding **<resource-config>** does not specify a value for the attribute.

Global parameters

You can specify **<param>** elements in every **<resource-config>**. These parameter values will either be available at runtime through the **SmooksResourceConfiguration** or, if not, they will be injected through the **@ConfigParam** annotation.

Global configuration parameters are defined in one place. Every runtime component can access them by using the **ExecutionContext**.

1.30. GLOBAL CONFIGURATION PARAMETERS

1. Global parameters are specified in a **<params>** element as shown:

```
<params>
    <param name="xyz.param1">param1-val</param>
</params>
```

2. Access the global parameters via the **ExecutionContext**:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"
    xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd"
    default-selector="order">

    <resource-config>
        <resource>com.acme.VisitorA</resource>
        ...
    </resource-config>

    <resource-config>
        <resource>com.acme.VisitorB</resource>
        ...
    </resource-config>

<smooks-resource-list>
```

1.31. DEFAULT PROPERTIES

Default properties can be set on the root element of a Smooks configuration which then applies them applied the resource configurations in the **smooks-conf.xml** file. If all of the resource configurations have the same selector value, you can specify a **default-selector=order**. This means you don't have to specify the selector on every resource configuration.

1.32. DEFAULT PROPERTIES EXAMPLE CONFIGURATION

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
    xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd"
    default-selector="order">
```

```
    <resource-config>
        <resource>com.acme.VisitorA</resource>
        ...
    </resource-config>

    <resource-config>
        <resource>com.acme.VisitorB</resource>
        ...
    </resource-config>

<smooks-resource-list>
```

1.33. DEFAULT PROPERTY OPTIONS

default-selector

This is applied to all of the resource-config elements in the Smooks configuration file if no other selector has been defined.

default-selector-namespace

This is the default selector namespace. It is used if no other namespace is defined.

default-target-profile

This is the default target profile. It is applied to all of the resources in the Smooks configuration file when no other target-profile has been defined.

default-condition-ref

This refers to a global condition by the conditions identifier. This condition is applied to resources that define an empty condition element (in other words, <condition/>) that does not reference a globally-defined condition.

1.34. FILTER SETTINGS

- To set filtering options, use the smooks-core configuration namespace. See the following example:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"
   xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd">
   <core:filterSettings type="SAX" defaultSerialization="true"
      terminateOnException="true" readerPoolSize="3"
closeSource="true"
      closeResult="true" rewriteEntities="true" />

      .. Other visitor configs etc...

</smooks-resource-list>
```

1.35. FILTER OPTIONS

type

This determines the type of processing model that will be used out of either SAX or DOM. (The default is DOM.)

defaultSerialization

This determines if default serialization should be switched on. The default value is **`true`**. Turning it on tells Smooks to locate a **`StreamResult`** (or **`DOMResult`**) in the result objects provided to the **`Smooks.filterSource`** method and to, by default, serialize all events to that result.

You can turn this behaviour off via the global configuration parameter or you can override it on a per-fragment basis by targeting a visitor implementation at that fragment that either takes ownership of the result writer (when using SAX filtering) or modifies the DOM (when using DOM filtering).

terminateOnException

Use this to determine whether an exception should terminate processing. The default setting is **`true`**.

closeSource

This closes source instance streams passed to the **`Smooks.filterSource`** method (the default is **`true`**). The exception here is **`System.in`**, which will never be closed.

closeResult

This closes result streams passed to the **`Smooks.filterSource`** method (the default is**`true`**). The exceptions here are **`System.out`** and **`System.err`**, which are never closed.

rewriteEntities

Use this to rewrite XML entities when reading and writing (default serialization) XML.

readerPoolSize

This sets the reader pool size. Some reader implementations are very expensive to create. Pooling reader instances (in other words, reusing them) can result in significant performance improvement, especially when processing a multitude of small messages. The default value for this setting is **`0`** (in other words, not pooled: a new reader instance is created for each message).

Configure this to be in line with your applications threading model.

CHAPTER 2. CONSUMING INPUT DATA

2.1. STREAM READERS

A *stream reader* is a class that implements the **`XMLReader`** interface (or the **`SmooksXMLReader`** interface). Smooks uses a stream reader to generate a stream of SAX events from the source message data stream. **`XMLReaderFactory.createXMLReader()`** is the default XMLReader. It can be configured to read non-XML data sources by configuring a specialist XML reader.

NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

2.2. XMLREADER CONFIGURATIONS

This is an example of how to configure the XML to use handlers, features and parameters:

```
<reader class="com.acme.ZZZZReader">
    <handlers>
        <handler class="com.X" />
        <handler class="com.Y" />
    </handlers>
    <features>
        <setOn feature="http://a" />
        <setOn feature="http://b" />
        <setOff feature="http://c" />
        <setOff feature="http://d" />
    </features>
    <params>
        <param name="param1">val1</param>
        <param name="param2">val2</param>
    </params>
</reader>
```

2.3. SETTING FEATURES ON THE XML READER

- By default, Smooks reads XML data. To set features on the default XML reader, omit the class name from the configuration:

```
<reader>
    <features>
        <setOn feature="http://a" />
        <setOn feature="http://b" />
        <setOff feature="http://c" />
        <setOff feature="http://d" />
    </features>
</reader>
```

2.4. CONFIGURING THE CSV READER

1. Use the http://www.milyn.org/xsd/smooks/csv-1.2.xsd configuration namespace to configure the reader.

   Here is a basic configuration:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <!--
    Configure the CSV to parse the message into a stream of SAX
events.
    -->
    <csv:reader fields="firstname,lastname,gender,age,country"
separator="|" quote="'" skipLines="1" />

</smooks-resource-list>
```

2. You will see the following event stream:

```
<csv-set>
    <csv-record>
        <firstname>Tom</firstname>
        <lastname>Fennelly</lastname>
        <gender>Male</gender>
        <age>21</age>
        <country>Ireland</country>
    </csv-record>
    <csv-record>
        <firstname>Tom</firstname>
        <lastname>Fennelly</lastname>
        <gender>Male</gender>
        <age>21</age>
        <country>Ireland</country>
    </csv-record>
</csv-set>
```

2.5. DEFINING CONFIGURATIONS

1. To define fields in XML configurations you must use a comma-separated list of names in the fields attribute.

2. Make sure the field names follow the same naming rules as XML element names:

3. - they can contain letters, numbers, and other characters
   - they cannot start with a number or punctuation character
   - they cannot start with the letters xml (or XML or Xml, etc)
   - they cannot contain spaces

4. Set the rootElementName and recordElementName attributes so you can modify the csv-set and csv-record element names. The same rules apply for these names.

5. You can define string manipulation functions on a per-field basis. These functions are executed before the data is converted into SAX events. Define them after the field name, separating the two with a question mark:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="lastname?trim.capitalize,country?upper_case"
/>

</smooks-resource-list>
```

6. To get Smooks to ignore fields in a CSV record, you must specify the $ignore$ token as the field's configuration value. Specify the number of fields to be ignored simply by following the $ignore$ token with a number (so use **`$ignore$3`** to ignore the next three fields.) Use **`$ignore$+`** to ignore all of the fields to the end of the CSV record.

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,$ignore$2,age,$ignore$+" />

</smooks-resource-list>
```

2.6. BINDING CSV RECORDS TO JAVA OBJECTS

1. Read the following to learn how to CSV records to Java objects. In this example, we will use CSV records for people:

```
Tom,Fennelly,Male,4,Ireland
Mike,Fennelly,Male,2,Ireland
```

2. Input this code to bind the record to a person:

```
public class Person {
    private String firstname;
    private String lastname;
    private String country;
    private Gender gender;
    private int age;
}

public enum Gender {
    Male,
    Female;
}
```

3. Input the following code and modify it to suit your task:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,lastname,gender,age,country">
        <!-- Note how the field names match the property names on
the Person class. -->
        <csv:listBinding beanId="people"
class="org.milyn.csv.Person" />
    </csv:reader>

</smooks-resource-list>
```

4. To execute the configuration, use this code:

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(csvStream), result);

List<Person> people = (List<Person>) result.getBean("people");
```

5. You can create Maps from the CSV record set:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:csv="http://www.milyn.org/xsd/smooks/csv-1.2.xsd">

    <csv:reader fields="firstname,lastname,gender,age,country">
        <csv:mapBinding beanId="people" class="org.milyn.csv.Person"
keyField="firstname" />
    </csv:reader>

</smooks-resource-list>
```

6. The configuration above produces a map of person instances, keyed to the firstname value of each person. This is how it is executed:

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(csvStream), result);

Map<String, Person> people = (Map<String, Person>)
result.getBean("people");
```

```
Person tom = people.get("Tom");
Person mike = people.get("Mike");
```

Virtual models are also supported, so you can define the class attribute as a **`java.util.Map`** and bind the CSV field values to map instances which are, in turn, added to a list or map.

2.7. CONFIGURING THE CSV READER FOR RECORD SETS

1. To configure a Smooks instance with a CSV reader to read a person record set, use the code below. It will bind the records to a list of person instances.

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new
CSVReaderConfigurator("firstname,lastname,gender,age,country")
                  .setBinding(new CSVBinding("people",
Person.class, CSVBindingType.LIST)));

JavaResult result = new JavaResult();
smooks.filterSource(new StreamSource(csvReader), result);

List<Person> people = (List<Person>) result.getBean("people");
```

NOTE

You can also optionally configure the Java Bean. The Smooks instance could instead (or additionally) be configured programmatically to use other visitor implementations to process the CSV record set.

2. To bind the CSV's records to a list or map of a Java type that reflects the data in your CSV records, use the **`CSVListBinder`** or **`CSVMapBinder`** classes:

```
// Note: The binder instance should be cached and reused...
CSVListBinder binder = new
CSVListBinder("firstname,lastname,gender,age,country",
Person.class);

List<Person> people = binder.bind(csvStream);

CSVMapBinder:

// Note: The binder instance should be cached and reused...
CSVMapBinder binder = new
CSVMapBinder("firstname,lastname,gender,age,country", Person.class,
"firstname");

Map<String, Person> people = binder.bind(csvStream);
```

If you need more control over the binding process, revert back to using the lower-level APIs.

2.8. CONFIGURING THE FIXED-LENGTH READER

1. To configure the fixed-length reader, modify the http://www.milyn.org/xsd/smooks/fixed-length-1.3.xsd configuration namespace as shown below:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-
1.3.xsd">

    <!--
    Configure the Fixed length to parse the message into a stream of
SAX events.
    -->
    <fl:reader
fields="firstname[10],lastname[10],gender[1],age[2],country[2]"
skipLines="1" />

</smooks-resource-list>
```

Here is an example input file:

```
#HEADER
Tom       Fennelly  M 21 IE
Maurice  Zeijen     M 27 NL
```

Here is the event stream that will be generated:

```
<set>
    <record>
        <firstname>Tom</firstname>
        <lastname>Fennelly</lastname>
        <gender>M</gender>
        <age>21</age>
        <country>IE</country>
    </record>
    <record>
        <firstname>Maurice</firstname>
        <lastname>Zeijen</lastname>
        <gender>M</gender>
        <age>27</age>
        <country>NL</country>
    </record>
</set>
```

2. Define the string manipulation functions as shown below:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-
```

```
1.3.xsd">

    <!--
    Configure the fixed length reader to parse the message into a
stream of SAX events.
    -->
    <fl:reader fields="firstname[10]?
trim,lastname[10]trim.capitalize,gender[1],age[2],country[2]"
skipLines="1" />

</smooks-resource-list>
```

3. You can also ignore these fields if you choose:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-
1.3.xsd">

    <fl:reader fields="firstname,$ignore$[2],age,$ignore$[10]" />

</smooks-resource-list>
```

2.9. CONFIGURING FIXED-LENGTH RECORDS

1. To bind fixed-length records to a person, see the configuration below. In this example we will use these sample records:

```
Tom Fennelly   M 21 IE
Maurice Zeijen M 27 NL
```

This is how you bind them to a person:

```
public class Person {
    private String firstname;
    private String lastname;
    private String country;
    private String gender;
    private int age;
}
```

2. Configure the records so they look like this:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"  xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-
1.3.xsd">

    <fl:reader fields="firstname[10]?trim,lastname[10]?
trim,gender[1],age[3]?trim,country[2]">
        <!-- Note how the field names match the property names on
```

```
the Person class. -->
        <fl:listBinding BeanId="people"
class="org.milyn.fixedlength.Person" />
    </fl:reader>

</smooks-resource-list>
```

3. Execute it as shown:

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(fixedLengthStream), result);

List<Person> people = (List<Person>) result.getBean("people");
```

4. Optionally, use this configuration to create maps from the fixed-length record set:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"  xmlns:fl="http://www.milyn.org/xsd/smooks/fixed-length-
1.3.xsd">

    <fl:reader fields="firstname[10]?trim,lastname[10]?
trim,gender[1],age[3]?trim,country[2]">
        <fl:mapBinding BeanId="people"
class="org.milyn.fixedlength.Person" keyField="firstname" />
    </fl:reader>

</smooks-resource-list>
```

5. This is how you execute the map of person instances that is produced:

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(fixedLengthStream), result);

Map<String, Person> people = (Map<String, Person>)
result.getBean("people");

Person tom = people.get("Tom");
Person mike = people.get("Maurice");
```

Virtual Models are also supported, so you can define the class attribute as a java.util.Map and bind the fixed-length field values to map instances, which are in turn added to a list or a map.

2.10. CONFIGURING THE FIXED-LENGTH READER PROGRAMMATICALLY

1. Use this code to configure the fixed-length reader to read a person record set, binding the record set into a list of person instances:

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new
FixedLengthReaderConfigurator("firstname[10]?trim,lastname[10]?
trim,gender[1],age[3]?trim,country[2]")
                  .setBinding(new FixedLengthBinding("people",
Person.class, FixedLengthBindingType.LIST)));

JavaResult result = new JavaResult();
smooks.filterSource(new StreamSource(fixedLengthStream), result);

List<Person> people = (List<Person>) result.getBean("people");
```

Configuring the Java binding is not mandatory. You can instead programmatically configure the Smooks instance to use other visitor implementations to carry out various forms of processing on the fixed-length record set.

2. To bind fixed-length records directly to a list or map of a Java type that reflects the data in your fixed-length records, use either the FixedLengthListBinder or the FixedLengthMapBinder classes:

```
// Note: The binder instance should be cached and reused...
FixedLengthListBinder binder = new
FixedLengthListBinder("firstname[10]?trim,lastname[10]?
trim,gender[1],age[3]?trim,country[2]", Person.class);

List<Person> people = binder.bind(fixedLengthStream);

FixedLengthMapBinder:

// Note: The binder instance should be cached and reused...
FixedLengthMapBinder binder = new
FixedLengthMapBinder("firstname[10]?trim,lastname[10]?
trim,gender[1],age[3]?trim,country[2]", Person.class, "firstname");

Map<String, Person> people = binder.bind(fixedLengthStream);
```

If you need more control over the binding process, revert back to the lower level APIs.

2.11. EDI PROCESSING

1. To utilize EDI processing in Smooks, access the http://www.milyn.org/xsd/smooks/edi-1.2.xsd configuration namespace.

2. Modify this configuration to suit your needs:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:edi="http://www.milyn.org/xsd/smooks/edi-1.2.xsd">
 <!--
    Configure the EDI Reader to parse the message stream into a
stream of SAX events.
    -->
```

```
    <edi:reader mappingModel="edi-to-xml-order-mapping.xml"
validate="false"/>
</smooks-resource-list>
```

2.12. EDI PROCESSING TERMS

- mappingModel: This defines the EDI mapping model configuration for converting the EDI message stream to a stream of SAX events that can be processed by Smooks.
- validate: This attribute turns the data-type validation in the EDI Parser on and off. (Validation is on by default.) To avoid redundancy, turn data-type validation off on the EDI reader if the EDI data is being bound to a Java object model (using Java bindings a la jb:bean).

2.13. EDI TO SAX

The EDI to SAX event mapping process is based on a mapping model supplied to the EDI reader. (This model must always use the http://www.milyn.org/xsd/smooks/edi-1.2.xsd schema. From this schema, you can see that segment groups are supported, including groups within groups, repeating segments and repeating segment groups.)

The medi:segment element supports two optional attributes, minOccurs and maxOccurs. (There is a default value of one in each case.) Use these attributes to control the characteristics of a segment. A maxOccurs value of -1 indicates that the segment can repeat any number of times in that location of the (unbound) EDI message.

You can add segment groups by using the segmentGroup element. A segment group is matched to the first segment in the group. They can contain nested segmentGroup elements, but the first element in a segmentGroup must be a segment. segmentGroup elements support minOccurs and maxOccurs. They also support an optional xmlTag attribute which, if present, will result in the XML generated by a matched segment group to be inserted into an element that has the name of the xmlTag attribute value.

2.14. EDI TO SAX EVENT MAPPING

When mapping EDI to SAX events, segments are matched in either of these ways:

- by an exact match on the segment code (segcode).
- by a regex pattern match on the full segment, where the segcode attribute defines the regex pattern (for instance, **`segcode="1A\*a.*"`**).
- required: field, component and sub-component configurations support a "required" attribute, which flags that field, component or sub-component as requiring a value.
- by default, values are not required (fields, components and sub-components).
- truncatable: segment, field and component configurations support a "truncatable" attribute. For a segment, this means that parser errors will not be generated when that segment does not specify trailing fields that are not "required" (see "required" attribute above). Likewise for fields/components and components/sub-components.
- By default, segments, fields, and components are not truncatable.

So, a field, component or a sub-component can be present in a message in one of the following states:

- present with a value (**`required="true"`**)

- present without a value (**`required="false"`**)
- absent (**`required="false" and truncatable="true"`**)

2.15. SEGMENT DEFINITIONS

It is possible to reuse segment definitions. Below is a configuration that demonstrates the importation feature:

```
<?xml version="1.0" encoding="UTF-8"?>
<medi:edimap xmlns:medi="http://www.milyn.org/schema/edi-message-mapping-
1.2.xsd">

    <medi:import truncatableSegments="true" truncatableFields="true"
truncatableComponents="true" resource="example/edi-segment-definition.xml"
namespace="def"/>

    <medi:description name="DVD Order" version="1.0"/>

    <medi:delimiters segment="&#10;" field="*" component="^" sub-
component="~" escape="?"/>

    <medi:segments xmltag="Order">
        <medi:segment minOccurs="0" maxOccurs="1" segref="def:HDR"
segcode="HDR" xmltag="header"/>
        <medi:segment minOccurs="0" maxOccurs="1" segref="def:CUS"
segcode="CUS" xmltag="customer-details"/>
        <medi:segment minOccurs="0" maxOccurs="-1" segref="def:ORD"
segcode="ORD" xmltag="order-item"/>
    </medi:segments>

</medi:edimap>
```

2.16. SEGMENT TERMS

Segments and segments containing child segments can be separated into another file for easier future reuse.

- segref: This contains a namespace:name referencing the segment to import.
- truncatableSegments: This overrides the truncatableSegments specified in the imported resource mapping file.
- truncatableFields: This overrides the truncatableFields specified in the imported resource mapping file.
- truncatableComponents: This overrides the truncatableComponents specified in the imported resource mapping file.

2.17. THE TYPE ATTRIBUTE

The example below demonstrates support for the type attribute.

```
<medi:edimap xmlns:medi="http://www.milyn.org/schema/edi-message-mapping-
```

```
1.2.xsd">

    <medi:description name="Segment Definition DVD Order" version="1.0"/>

    <medi:delimiters segment="&#10;" field="*" component="^" sub-
component="~" escape="?"/>

    <medi:segments xmltag="Order">

        <medi:segment segcode="HDR" xmltag="header">
            <medi:field xmltag="order-id"/>
            <medi:field xmltag="status-code" type="Integer"/>
            <medi:field xmltag="net-amount" type="BigDecimal"/>
            <medi:field xmltag="total-amount" type="BigDecimal"/>
            <medi:field xmltag="tax" type="BigDecimal"/>
            <medi:field xmltag="date" type="Date"
typeParameters="format=yyyyHHmm"/>
        </medi:segment>

    </medi:segments>

</medi:edimap>
```

You can use type system for different things, including:

- field validation
- Edifact Java Compilation

2.18. THE EDIREADERCONFIGURATOR

- Use the **EDIReaderConfigurator** to programmatically configure the Smooks instance to use the EDIReader as shown in the code below:

```
Smooks smooks = new Smooks();

// Create and initialise the Smooks config for the parser...
smooks.setReaderConfig(new
EDIReaderConfigurator("/edi/models/invoice.xml"));

// Use the smooks as normal
smooks.filterSource(....);
```

2.19. THE EDIFACT JAVA COMPILER

The **Edifact Java Compiler** simplifies the process of going from EDI to Java. It generates the following:

- a Java object model for a given EDI mapping model.
- a Smooks Java binding configuration to populate the Java Object model from an instance of the EDI message described by the EDI mapping model.
- a factory class to use the **Edifact Java Compiler** to bind EDI data to the Java object model.

2.20. EDIFACT JAVA COMPILER EXAMPLE

The **Edifact Java Compiler** allows you to write simple Java code such as the following:

```
// Create an instance of the EJC generated Factory class.  This should
normally be cached and reused...
OrderFactory orderFactory = OrderFactory.getInstance();

// Bind the EDI message stream data into the EJC generated Order model...
Order order = orderFactory.fromEDI(ediStream);

// Process the order data...
Header header = order.getHeader();
Name name = header.getCustomerDetails().getName();
List<OrderItem> orderItems = order.getOrderItems();
```

2.21. EXECUTING THE EDIFACT JAVA COMPILER

- To execute the **Edifact Java Compiler** through **`Maven`**, add the plug-in in your POM file:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.milyn</groupId>
            <artifactId>maven-ejc-plugin</artifactId>
            <version>1.2</version>
            <configuration>
                <ediMappingFile>edi-model.xml</ediMappingFile>
                <packageName>com.acme.order.model</packageName>
            </configuration>
            <executions>
                <execution><goals><goal>generate</goal></goals>
</execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

2.22. MAVEN PLUG-IN PARAMETERS FOR THE EDIFACT JAVA COMPILER

The plug-in has three configuration parameters:

- ediMappingFile: the path to the EDI mapping model file within the **`Maven`** project. (It is optional. The default is **`src/main/resources/edi-model.xml`**).
- packageName:the Java package the generated Java artifacts are placed into (the Java object model and the factory class).
- destDir: the directory in which the generated artifacts are created and compiled. (This is optional. The default is **`target/ejc`**).

2.23. EXECUTING THE EDIFACT JAVA COMPILER WITH ANT

- Create and execute the EJC task as shown below:

```
<target name="ejc">

    <taskdef resource="org/milyn/ejc/ant/anttasks.properties">
        <classpath><fileset dir="/smooks-1.2/lib" includes="*.jar"/>
</classpath>
    </taskdef>

    <ejc edimappingmodel="src/main/resources/edi-model.xml"
         destdir="src/main/java"
         packagename="com.acme.order.model"/>

    <!-- Ant as usual from here on... compile and jar the source...
-->

</target>
```

2.24. UN/EDIFACT MESSAGE INTERCHANGES

The easiest way to learn more about the **Edifact Java Compiler** is to check out the EJC example, UN/EDIFACT.

Smooks provides out-of-the-box support for UN/EDIFACT message interchanges by way of these means:

- pre-generated EDI mapping models generated from the official UN/EDIFACT message definition ZIP directories. These allow you to convert a UN/EDIFACT message interchange into a more readily consumable XML format.
- pre-generated Java bindings for easy reading and writing of UN/EDIFACT interchanges using pure Java

2.25. USING UN/EDIFACT INTERCHANGES WITH THE EDI:READER

- Set the http://www.milyn.org/xsd/smooks/unedifact-1.4.xsd namespace like this:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:unedifact="http://www.milyn.org/xsd/smooks/unedifact-
1.4.xsd">

    <unedifact:reader
mappingModel="urn:org.milyn.edi.unedifact:d03b-mapping:v1.4"
ignoreNewLines="true" />

</smooks-resource-list>
```

The mappingModel attribute defines an *URN* that refers to the mapping model ZIP set's **Maven** artifact, which is used by the reader.

2.26. CONFIGURING SMOOKS TO CONSUME A UN/EDIFACT INTERCHANGE

1. To programmatically configure Smooks to consume a UN/EDIFACT interchange (via, for instance, an UNEdifactReaderConfigurator), use the code below:

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new
UNEdifactReaderConfigurator("urn:org.milyn.edi.unedifact:d03b-
mapping:v1.4"));
```

2. Insert the following on the containing application's classpath:

   - the requisite EDI mapping models
   - the Smooks EDI cartridge

3. There may be some **Maven** dependancies your configuration will require. See the example below:

```
<dependency>
    <groupId>org.milyn</groupId>
    <artifactId>milyn-smooks-edi</artifactId>
    <version>1.4</version>
</dependency>

<!-- Required Mapping Models -->
<dependency>
    <groupId>org.milyn.edi.unedifact</groupId>
    <artifactId>d93a-mapping</artifactId>
    <version>v1.4</version>
</dependency>
<dependency>
    <groupId>org.milyn.edi.unedifact</groupId>
    <artifactId>d03b-mapping</artifactId>
    <version>v1.4</version>
</dependency>
```

4. Once an application has added an EDI mapping model ZIP set to its classpath, you can configure **Smooks** to use this model by simply referencing the **Maven** artifact using a URN as the unedifact:reader configuration's mappingModel attribute value:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:unedifact="http://www.milyn.org/xsd/smooks/unedifact-
1.4.xsd">

    <unedifact:reader
mappingModel="urn:org.milyn.edi.unedifact:d03b-mapping:v1.4"
ignoreNewLines="true" />

</smooks-resource-list>
```

2.27. THE MAPPINGMODEL

The *mappingModel* attribute can define multiple, comma-separated EDI Mapping Models URNs . By doing so, it facilitates the UN/EDIFACT reader process interchanges which deal with multiple UN/EDIFACT messages defined in different directories.

Mapping model ZIP sets are available for all of the UN/EDIFACT directories. Obtain them from the **MavenSNAPSHOT** and **Central** repositories and add them to your application by using standard **Maven** dependency management.

2.28. CONFIGURING THE MAPPINGMODEL

1. To add the D93A mapping model ZIP set to your application classpath, set the following dependency to your application's POM file:

```
<!-- The mapping model sip set for the D93A directory... -->
<dependency>
    <groupId>org.milyn.edi.unedifact</groupId>
    <artifactId>d93a-mapping</artifactId>
    <version>v1.4</version>
</dependency>
```

2. Configure Smooks to use this ZIP set by adding the **unedifact:reader** configuration to your Smooks configuration as shown below:

```
<unedifact:reader mappingModel="urn:org.milyn.edi.unedifact:d93a-
mapping:v1.4" />
```

   Note how you configure the reader using a URN derived from the **Maven** artifac's dependency information.

3. You can also add multiple mapping model ZIP sets to your application's classpath. To do so, add all of them to your **unedifact:reader** configuration by comma-separating the URNs.

4. Pre-generated Java binding model sets are provided with the tool (there is one per mapping model ZIP set). Use these to process UN/EDIFACT interchanges using a very simple, generated factory class.

2.29. PROCESSING A D03B UN/EDIFACT MESSAGE INTERCHANGE

1. To process a D03B UN/EDIFACT message interchange, follow the example below:

```
Reading:

// Create an instance of the EJC generated factory class... cache
this and reuse !!!
D03BInterchangeFactory factory =
D03BInterchangeFactory.getInstance();

// Deserialize the UN/EDIFACT interchange stream to Java...
UNEdifactInterchange interchange =
factory.fromUNEdifact(ediInStream);
```

```
// Need to test which interchange syntax version.  Supports v4.1 at
the moment...
if(interchange instanceof UNEdifactInterchange41) {
    UNEdifactInterchange41 interchange41 = (UNEdifactInterchange41)
interchange;

    for(UNEdifactMessage41 message : interchange41.getMessages()) {
        // Process the messages...

        Object messageObj = message.getMessage();

        if(messageObj instanceof Invoic) {
            // It's an INVOIC message....
            Invoic invoic = (Invoic) messageObj;
            ItemDescription itemDescription =
invoic.getItemDescription();
            // etc etc....
        } else if(messageObj instanceof Cuscar) {
            // It's a CUSCAR message...
        } else if(etc etc etc...) {
            // etc etc etc...
        }
    }
}


Writing:

factory.toUNEdifact(interchange, ediOutStream);
```

2. Use **Maven** to add the ability to process a D03B message interchange by adding the binding dependency for that directory (you can also use pre-generated UN/EDIFACT Java object models distributed via the **MavenSNAPSHOT** and **Central** repositories):

```
<dependency>
    <groupId>org.milyn.edi.unedifact</groupId>
    <artifactId>d03b-binding</artifactId>
    <version>v1.4</version>
</dependency>
```

2.30. PROCESSING JSON DATA

1. To process JSON data, you must configure a JSON reader:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

    <json:reader/>

</smooks-resource-list>
```

2. Set the XML names of the root, document and array elements by using the following configuration options:

   - rootName: this is the name of the root element. The default is yaml.

   - elementName: this is the name of a sequence element. The default is element.

3. You may wish to use characters in the key name that are not allowed in the XML element name. The reader offers multiple solutions to this problem. It can search and replace white spaces, illegal characters and the number in key names that start with a number. You can also use it to replace one key name with a completely different one. The following sample code shows you how to do this:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

    <json:reader keyWhitspaceReplacement="_" keyPrefixOnNumeric="n"
illegalElementNameCharReplacement=".">
        <json:keyMap>
            <json:key from="some key">someKey</json:key>
            <json:key from="some&key" to="someAndKey" />
        </json:keyMap>
    </json:reader>

</smooks-resource-list>
```

   - keyWhitspaceReplacement: this is the replacement character for white spaces in a JSON map key. By default this is not defined, so the reader does not automatically search for white spaces.

NOTE

Note that there is no **e** between **Whit** and **space** in the keyWhitspaceReplacement field name.

   - keyPrefixOnNumeric: this is the prefix character to add if the JSON node name starts with a number. By default, this is not defined, so the reader does not search for element names that start with a number.

   - illegalElementNameCharReplacement: if illegal characters are encountered in a JSON element name then they are replaced with this value.

4. You can also configure these optional settings:

5. - nullValueReplacement: this is the replacement string for JSON null values. The default is an empty string.

   - encoding: this is the default encoding of any JSON message InputStream processed by the reader. The default encoding is UTF-8.

NOTE

This feature is deprecated. Instead, you should now manage the JSON streamsource character encoding by supplying a **`java.io.Reader`** to the **`Smooks.filterSource()`** method.

6. To configure Smooks programmatically to read a JSON configuration, use the **`JSONReaderConfigurator`**class:

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new JSONReaderConfigurator()
        .setRootName("root")
        .setArrayElementName("e"));

// Use Smooks as normal...
```

2.31. USING CHARACTERS NOT ALLOWED IN XML WHEN PROCESSING JSON DATA

To use characters in the key name that are not allowed in the XML element name, use the reader to search and replace white spaces, illegal characters and the number in key names that start with a number. You can also use it to replace one key name with a completely different one. The following sample code shows you how to do this:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:json="http://www.milyn.org/xsd/smooks/json-1.1.xsd">

    <json:reader keyWhitspaceReplacement="_" keyPrefixOnNumeric="n"
illegalElementNameCharReplacement=".">
        <json:keyMap>
            <json:key from="some key">someKey</json:key>
            <json:key from="some&key" to="someAndKey" />
        </json:keyMap>
    </json:reader>

</smooks-resource-list>
```

- keyWhitspaceReplacement: this is the replacement character for white spaces in a JSON map key. By default this is not defined, so the reader does not automatically search for white spaces.
- keyPrefixOnNumeric: this is the prefix character to add if the JSON node name starts with a number. By default, this is not defined, so the reader does not search for element names that start with a number.
- illegalElementNameCharReplacement: if illegal characters are encountered in a JSON element name then they are replaced with this value.

These settings are optional:

- nullValueReplacement: this is the replacement string for JSON null values. The default is an empty string.

- encoding: this is the default encoding of any JSON message InputStream processed by the reader. The default encoding is UTF-8.

NOTE

This feature is deprecated. Instead, you should now manage the JSON streamsource character encoding by supplying a **`java.io.Reader`** to the **`Smooks.filterSource()`** method.

2.32. CONFIGURING YAML STREAMS

Procedure 2.1. Task

1. Configure your reader to process YAML files as shown:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:yaml="http://www.milyn.org/xsd/smooks/yaml-1.4.xsd">

    <yaml:reader/>

</smooks-resource-list>
```

2. Configure the YAML stream to contain multiple documents. The reader handles this by adding a document element as a child of the root element. An XML-serialized YAML stream with one empty YAML document looks like this:

```
<yaml>
    <document>
    </document>
</yaml>
```

3. Configure Smooks programmatically to read a YAML configuration by exploiting the **`YamlReaderConfigurator`** class:

```
Smooks smooks = new Smooks();

smooks.setReaderConfig(new YamlReaderConfigurator()
        .setRootName("root")
        .setDocumentName("doc")
        .setArrayElementName("e"))
        .setAliasStrategy(AliasStrategy.REFER_RESOLVE)
        .setAnchorAttributeName("anchor")
        .setAliasAttributeName("alias");

// Use Smooks as normal...
```

2.33. SUPPORTED RESULT TYPES

Smooks can work with standard **`JDK StreamResult`** and **`DOMResult`** result types, as well as these specialist ones:

- **`JavaResult`**: use this result type to capture the contents of the Smooks Java Bean context.
- **`ValidationResult`**: use this result type to capture outputs.
- Simple Result type: use this when writing tests. This is a **`StreamResult`** extension wrapping a **`StringWriter`**.

2.34. USING CHARACTERS NOT ALLOWED IN XML WHEN PROCESSING YAML DATA

- You can use characters in the key name that are not allowed in the XML element name. The reader offers multiple solutions to this problem. It can search and replace white spaces, illegal characters and the number in key names that start with a number. You can configure it to replace one key name with a completely different one, as shown below:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:yaml="http://www.milyn.org/xsd/smooks/yaml-1.4.xsd">

    <yaml:reader keyWhitspaceReplacement="_" keyPrefixOnNumeric="n"
illegalElementNameCharReplacement=".">
        <yaml:keyMap>
            <yaml:key from="some key">someKey</yaml:key>
            <yaml:key from="some&key" to="someAndKey" />
        </yaml:keyMap>
    </yaml:reader>

</smooks-resource-list>
```

2.35. OPTIONS FOR REPLACING XML IN YAML

- keyWhitspaceReplacement: This is the replacement character for white spaces in a YAML map key. By default this not defined.
- keyPrefixOnNumeric: Add this prefix if the YAML node name starts with a number. By default this is not defined.
- illegalElementNameCharReplacement: If illegal characters are encountered in a YAML element name, they are replaced with this value. By default this is not defined.

2.36. ANCHORS AND ALIASES IN YAML

The YAML reader can handle *anchors* and *aliases* via three different strategies. Define your strategy of choice via the aliasStrategy configuration option. This option can have one of the following values:

- REFER: The reader creates reference attributes on the element that has an anchor or an alias. The element with the anchor obtains the id attribute containing the name from the anchor as the attribute value. The element with the alias gets the ref attribute also containing the name of the anchor as the attribute value. You can define the anchor and alias attribute names by setting the anchorAttributeName and aliasAttributeName properties.
- RESOLVE: The reader resolves the value or the data structure of an anchor when its alias is encountered. This means that the SAX events of the anchor are repeated as child events of the alias element. When a YAML document contains a lot of anchors or anchors and a substantial

data structure this can lead to memory problems.

- REFER_RESOLVE: This is a combination of REFER and RESOLVE. The anchor and alias attributes are set but the anchor value or data structure is also resolved. This option is useful when the name of the anchor has a business meaning.

The YAML reader uses the REFER strategy by default.

2.37. JAVA OBJECT GRAPH TRANSFORMATION

1. Smooks can transform one *Java object graph* into another. To do this, it uses the SAX processing model, which means no intermediate object model is constructed. Instead, the source Java object graph is turned directly into a stream of SAX events, which are used to populate the target Java object graph.

   If you use the HTML Smooks Report Generator tool, you will see that the event stream produced by the source object model is as follows:

```
<example.srcmodel.Order>
    <header>
        <customerNumber>
            </customerNumber>
           <customerName>
       </customerName>
    </header>
    <orderItems>
        <example.srcmodel.OrderItem>
            <productId>
           </productId>
            <quantity>
            </quantity>
            <price>
            </price>
        </example.srcmodel.OrderItem>
    </orderItems>
</example.srcmodel.Order>
```

2. Aim the Smooks Java bean resources at this event stream. The Smooks configuration for performing this transformation (*smooks-config.xml*) is as follows:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd">

    <jb:bean BeanId="lineOrder" class="example.trgmodel.LineOrder"
createOnElement="example.srcmodel.Order">
        <jb:wiring property="lineItems" BeanIdRef="lineItems" />
        <jb:value property="customerId" data="header/customerNumber"
/>
        <jb:value property="customerName" data="header/customerName"
/>
    </jb:bean>

    <jb:bean BeanId="lineItems" class="example.trgmodel.LineItem[]"
```

```
createOnElement="orderItems">
        <jb:wiring BeanIdRef="lineItem" />
    </jb:bean>


    <jb:bean BeanId="lineItem" class="example.trgmodel.LineItem"
createOnElement="example.srcmodel.OrderItem">
        <jb:value property="productCode"
data="example.srcmodel.OrderItem/productId" />
        <jb:value property="unitQuantity"
data="example.srcmodel.OrderItem/quantity" />
        <jb:value property="unitPrice"
data="example.srcmodel.OrderItem/price" />
    </jb:bean>

</smooks-resource-list>
```

3. The source object model is provided to Smooks via a **`org.milyn.delivery.JavaSource`** object. Create this object by passing the constructor the source model's root object. The resulting Java Source object is used in the **`Smooks#filter`** method. Here is the resulting code:

```
protected LineOrder runSmooksTransform(Order srcOrder) throws
IOException, SAXException {
    Smooks smooks = new Smooks("smooks-config.xml");
    ExecutionContext executionContext =
smooks.createExecutionContext();

    // Transform the source Order to the target LineOrder via a
    // JavaSource and JavaResult instance...
    JavaSource source = new JavaSource(srcOrder);
    JavaResult result = new JavaResult();

    // Configure the execution context to generate a report...
    executionContext.setEventListener(new
HtmlReportGenerator("target/report/report.html"));

    smooks.filterSource(executionContext, source, result);

    return (LineOrder) result.getBean("lineOrder");
}
```

2.38. STRING MANIPULATION ON INPUT DATA

The CSV and fixed-length readers allow you to execute string manipulation functions on the input data before the data is converted into SAX events. The following functions are available:

- upper_case: this returns the upper case version of the string.
- lower_case: this returns the lower case version of the string.
- cap_first: this returns the string with the very first word capitalized.
- uncap_first: this returns the string with the very first word un-capitalized. It is the opposite of cap_first.

- capitalize: this returns the string with all words capitalized.
- trim: this returns the string without leading and trailing white-spaces.
- left_trim: this returns the string without leading white-spaces.
- right_trim: this returns the string without trailing white-spaces.

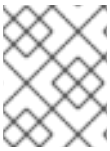
You can chain functions via the point separator. Here is an example: **`trim.upper_case`**

How you define the functions per field depends on the reader you are using.

CHAPTER 3. VALIDATION

3.1. RULES IN SMOOKS

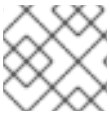
In Smooks, *rules* are a general concept, not something specific to a particular cartridge.



NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

You can configure and reference a *RuleProvider* from other components.



NOTE

The only cartridge that uses rules functionality is the validation cartridge.

Rules are centrally defined through *ruleBases*. A single Smooks configuration can refer to many ruleBase definitions. A rulesBase configuration has a name, a rule **`src`** and a rule provider.

The format of the rule source is entirely dependent on the provider implementation. The only requirement is that the individual rules be uniquely named (within the context of a single source) so they can be referenced.

3.2. CONFIGURING RULES IN SMOOKS

Here is an example ruleBase configuration:

```
<?xml version="1.0"?>
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd">

    <rules:ruleBases>
        <rules:ruleBase name="regexAddressing"
src="/org/milyn/validation/address.properties"
provider="org.milyn.rules.regex.RegexProvider" />
        <rules:ruleBase name="order"
src="/org/milyn/validation/order/rules/order-rules.csv"
provider="org.milyn.rules.mvel.MVELProvider"/>
    </rules:ruleBases>

</smooks-resource-list>
```

3.3. MANDATORY CONFIGURATIONS FOR THE RULES:RULEBASE CONFIGURATION ELEMENT

- name: this is used by other components to refer to this rule.
- src: this can be a file or anything else that is meaningful to the RuleProvider.

- provider: This is the actual provider implementation. In the configuration above, there is one RuleProvider that uses regular expressions but you can specify multiple ruleBase elements and have as many RuleProviders as you need.

3.4. RULE PROVIDERS

Rule providers implement the **`org.milyn.rules.RuleProvider`** interface.

Smooks comes pre-configured to support two RuleProvider implementations:

- RegexProvider
- MVELProvider

You can also create your own RuleProvider implementations.

3.5. THE REGEXPROVIDER

The *RegexProvider* utilises regular expressions. It allows you to define low-level rules specific to the format of selected data fields in the message being filtered. For example, it may be applied to a particular field to validate the syntax to make sure the right e-mail address is being used.

3.6. CONFIGURING A REGEX RULEBASE

1. Use this example code to configure a Regex ruleBase:

```
<?xml version="1.0"?>
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"  xmlns:rules="http://www.milyn.org/xsd/smooks/rules-
1.0.xsd">

    <rules:ruleBases>
        <rules:ruleBase name="customer"
src="/org/milyn/validation/order/rules/customer.properties"
provider="org.milyn.rules.regex.RegexProvider"/>
    </rules:ruleBases>

</smooks-resource-list>
```

2. Define the Regex expressions in a standard **`.properties`** file format. The following **`customer.properties`** Regex rule definition file example shows you how:

```
# Customer data rules...
customerId=[A-Z][0-9]{5}
customerName=[A-Z][a-z]*, [A-Z][a-z]
```

3.7. THE MVEL PROVIDER

The *MVEL* provider allows you to define rules as MVEL expressions. These expressions are executed over the contents of the Smooks Javabean context. You should to bind data from the message being filtered into Java objects in the Smooks bean context.

MVEL allows you to define more complex rules on message fragments. (Such as "Is the product in the targeted order item fragment within the age eligibility constraints of the customer specified in the order header details?")

3.8. CONFIGURING AN MVEL RULEBASE

1. To configure an MVEL ruleBase, see the code below:

```
<?xml version="1.0"?>
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"  xmlns:rules="http://www.milyn.org/xsd/smooks/rules-
1.0.xsd">

    <rules:ruleBases>
        <rules:ruleBase name="order"
src="/org/milyn/validation/order/rules/order-rules.csv"
provider="org.milyn.rules.mvel.MVELProvider"/>
    </rules:ruleBases>

</smooks-resource-list>
```

2. You must store your MVEL rules in CSV files. The easiest way to edit these files is through a spreadsheet application such as **LibreOffice Calc** or **Gnumeric**. Each rule record contains a rule name and an MVEL expression.

3. If you wish to create comment and header rows, prefix the first field with a hash (#) character.

3.9. THE SMOOKS VALIDATION CARTRIDGE

The Smooks validation cartridge works with the rules cartridge to provide *rules-based fragment validation*.

This allows you to perform detailed validation on message fragments. As with everything in Smooks, the validation functionality is available across all supported data formats. This means you can perform strong validation on not just XML data, but also on EDI, JSON, CSV and so on.

Validation configurations are defined by the http://www.milyn.org/xsd/smooks/validation-1.0.xsd configuration namespace.

Smooks supports a number of different *rule provider* types and these can all be used by the Validation Cartridge. Each of these provide a different level of validation but they are all configured in exactly the same way. The Smooks Validation Cartridge sees a rule provider as an abstract resource that it can aim at message fragments in order to validate them.

3.10. CONFIGURING VALIDATION RULES

To configure a validation rule you need to specify the following:

- executeOn: this is the fragment on which the rule is to be executed.
- excecuteOnNS: this is the fragment namespace to which the **`executeOn`** belongs.

- name: this is the name of the rule to be applied. This is a composite rule name that refers to a ruleBase and ruleName combination in a dot delimited format (in other words **`ruleBaseName.ruleName`**).
- onFail: this determines the severity of a failed match.

Here is a sample validation rule configuration:

```
    <validation:rule executeOn="order/header/email"
name="regexAddressing.email" onFail="ERROR" />
```

3.11. CONFIGURING VALIDATION EXCEPTIONS

1. You can set a maximum number of validation failures per Smooks filter operation. (An exception will be thrown if this maximum is exceeded.) Validations configured with OnFail.FATAL will always throw an exception and stop processing.

   To configure the maximum validation failures, add this code to your Smooks configuration:

   ```
   <params>
       <param name="validation.maxFails">5</param>
   </params>
   ```

2. The onFail attribute in the validation configuration specifies what action is to be taken. This determines how validation failures are to be reported. To utilize it, modify the following options to suit your needs:

   - OK: Use this to save the validation as "okay". By calling **`ValidationResults.getOks`** all validation warnings will be returned. This option is useful for content-based routing.
   - WARN: Use this to save the validation as a warning. By calling **`ValidationResults.getWarnings`** all validation warnings will be returned.
   - ERROR: Use this to save the validation as an error. By calling **`ValidationResults.getErrors`** you will return all validation errors.
   - FATAL: Use this to throw a ValidationException as soon as a validation failure occurs. If you call **`ValidationResults.getFatal`** you will see the fatal validation failure.

3.12. RULE BASES

- Use a composite rule name in the following format for a rule base:

  ```
  <ruleProviderName>.<ruleName>
  ```

  - ruleProviderName identifies the rule provider and maps to the **`name`** attribute in the **`ruleBase`** element.
  - ruleName identifies a specific rule the rule provider knows about. This could be a rule defined in the **`src`** file.

3.13. SMOOKS.FILTERSOURCE

The *Smooks.filterSource* captures the validation results. When the *filterSource* method returns, the ValidationResult instance will contain all validation data.

This code shows how to make Smooks perform message fragment validation:

```
ValidationResult validationResult = new ValidationResult();

smooks.filterSource(new StreamSource(messageInStream), new
StreamResult(messageOutStream), validationResult);

List<OnFailResult> errors = validationResult.getErrors();
List<OnFailResult> warnings = validationResult.getWarnings();
```

As you can see, individual warning and error validation results are made available from the **`ValidationResult`** object in the form of **`OnFailResult`** instances, each of which provide details about an individual failure.

The Validation Cartridge also allows you to specify localized messages relating to validation failures. You can define these messages in standard Java ResourceBundle files (which use the **`.properties`** format).

NOTE

The base name of the validation message bundle is derived from the rule source ("src") by dropping the rule source file extension and adding an extra folder named **`i18n`**. For example, if you have an MVEL ruleBase source of **`/org/milyn/validation/order/rules/order-rules.csv`**, the corresponding validation message bundle base name will be **`/org/milyn/validation/order/rules/i18n/order-rules`**.

3.14. THE VALIDATION CARTRIDGE AND MESSAGES

The validation cartridge lets you apply **FreeMarker** templates to the localized messages, allowing messages to contain contextual data from the bean context, as well as data about the actual rule failure. You must prefix **FreeMarker**-based messages with **`ftl:`** and reference the contextual data using standard **FreeMarker** notation. The beans from the bean context can be referenced directly, while you can refer to the RuleEvalResult and rule failure path through the **`ruleResult`** and **`path`** beans.

Here is an example that uses RegexProvider rules which shows how Smooks can be used to perform validation of message fragment data:

```
customerId=ftl:Invalid customer number '${ruleResult.text}' at '${path}'.
Customer number must match pattern '${ruleResult.pattern}'.
```

3.15. TYPES OF VALIDATION

Smooks performs two types of validation using two different kinds of validation rules:

- message field value/format validation using regular expressions defined in a **`.properties`** file RuleBase. This, for example, can be to validate a field as being a valid e-mail address.

- business rules validation using MVEL expressions defined in a **.csv** file RuleBase. This can, for example, be validating that the total price of an order item on an order (price * quantity) does not breach some predefined business rule.

3.16. RUNNING VALIDATION RULES

- To run validaton rules, go to the example root folder and execute:

  1. **mvn clean install**

  2. **mvn exec:java**

3.17. RULEBASE EXAMPLE

In this example, there is an XML message containing a collection of order items. (This functionality works similarly for all other data formats supported by Smooks.):

```
<Order>
    <header>
        <orderId>A188127</orderId>
        <username>user1</username>
        <name>
            <firstname>Bob</firstname>
            <lastname>Bobington</lastname>
        </name>
        <email>bb@awesomemail.com</email>
        <state>Queensland</state>
    </header>
    <order-item>
        <quantity>1</quantity>
        <productId>364b</productId>
        <title>A Great Movie</title>
        <price>29.95</price>
    </order-item>
    <order-item>
        <quantity>2</quantity>
        <productId>299</productId>
        <title>A Terrible Movie</title>
        <price>29.95</price>
    </order-item>
</Order>
```

3.18. MESSAGE DATA VALIDATION

1. When processing an order message, you should perform a number of validations. First, check that the supplied username follows a format of an upper case character, followed by five digits (for example, **S12345** or **G54321**). To perform this validation, you should use regular expression.

2. Next, check that the supplied e-mail address is in a valid format. Use a regular expression to check it.

3. Confirm that each order item's productId field follows a format of exactly three digits (such as **123**). Use a regular expression to do this.

4. Finally, you need to confirm that the total for each order item does not exceed 50.00 (price * quantity is not greater than 50.00). Perform this validation using an MVEL expression.

3.19. USING AN MVEL EXPRESSION

1. To use an MVEL expression on a rule set, divide the Regex rules and place them in two separate **`.properties`** files.

2. Drop these rules into the example **`rules`** directory.

3. Put the MVEL expression in a **`.csv`** file, also in the **`rules`** directory.

   The customer-related Regex rules that go in the **`customer.properties`** file look like this:

```
# Customer data rules...
customerId=[A-Z][0-9]{5}

# Email address...
email=^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$
```

4. Insert the product-related Regex rule in the **`product.properties`** file:

```
# Product data rules...
productId=[0-9]{3}
```

5. Insert the MVEL expression for performing the order item total check into the **`order-rules.csv`** file.

NOTE

The easiest way to edit a .csv file is through using a spreadsheet application like **LibreOffice Calc** or **Gnumeric**.

6. Create resource bundle **`.properties`** files for each of the rule source files.

NOTE

The names of these files are derived from the names of their corresponding rule files.

The message bundle for the rules defined in **`rules/customer.properties`** is located in the **`rules/i18n/customer.properties`** file:

```
customerId=ftl:Invalid customer number '${ruleResult.text}' at
'${path}'. Customer number must begin with an uppercase character,
followed by 5 digits.
email=ftl:Invalid email address '${ruleResult.text}' at '${path}'.
Email addresses match pattern '${ruleResult.pattern}'.
```

The message bundle for the rule defined in **`rules/product.properties`** is located in the **`rules/i18n/product.properties`** file:

```
# Product data rule messages...
productId=ftl:Invalid product ID '${ruleResult.text}' at '${path}'.
Product ID must match pattern '${ruleResult.pattern}'.
```

The message bundle for the rule defined in **rules/order-rules.csv** is located in the **rules/i18n/order-rules.properties** file:

```
# Order item rule messages. The "orderDetails" and "orderItem" beans
are populated by Smooks bindings
order_item_total=ftl:Order ${orderDetails.orderId} contains an order
item for product ${orderItem.productId} with a quantity of
${orderItem.quantity} and a unit price of ${orderItem.price}. This
exceeds the permitted per order item total.
```

7. Apply this validation to the rules:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"

xmlns:rules="http://www.milyn.org/xsd/smooks/rules-1.0.xsd"

xmlns:validation="http://www.milyn.org/xsd/smooks/validation-
1.0.xsd"

xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.xsd">

    <params>
        <!-- Generate a ValidationException if we get more than 5
validation failures... -->
        <param name="validation.maxFails">5</param>
    </params>

    <!-- Define the ruleBases that are used by the validation
rules... -->
    <rules:ruleBases>
        <!-- Field value rules using regex... -->
        <rules:ruleBase name="customer"
src="rules/customer.properties"
provider="org.milyn.rules.regex.RegexProvider"/>
        <rules:ruleBase name="product"
src="rules/product.properties"
provider="org.milyn.rules.regex.RegexProvider"/>

        <!-- Order business rules using MVEL expressions... -->
        <rules:ruleBase name="order" src="rules/order-rules.csv"
provider="org.milyn.rules.mvel.MVELProvider"/>
    </rules:ruleBases>

    <!-- Capture some data into the bean context - required by the
business rule validations... -->
    <jb:bean beanId="orderDetails" class="java.util.HashMap"
createOnElement="header">
        <jb:value data="header/*"/>
    </jb:bean>
```

```
    <jb:bean beanId="orderItem" class="java.util.HashMap"
createOnElement="order-item">
        <jb:value data="order-item/*"/>
    </jb:bean>

    <!-- Target validation rules... -->
    <validation:rule executeOn="header/username"
name="customer.customerId" onFail="ERROR"/>
    <validation:rule executeOn="email" name="customer.email"
onFail="WARN"/>
    <validation:rule executeOn="order-item/productId"
name="product.productId" onFail="ERROR"/>

    <validation:rule executeOn="order-item"
name="order.order_item_total" onFail="ERROR"/>

</smooks-resource-list>
```

8. Execute from the example's **`Main`** class using this code:

```
protected static ValidationResult runSmooks(final String messageIn)
throws IOException, SAXException, SmooksException {
    // Instantiate Smooks with the config...
    final Smooks smooks = new Smooks("smooks-config.xml");

    try {
        // Create an exec context - no profiles....
        final ExecutionContext executionContext =
smooks.createExecutionContext();
        final ValidationResult validationResult = new
ValidationResult();

        // Configure the execution context to generate a report...
        executionContext.setEventListener(new
HtmlReportGenerator("target/report/report.html"));

        // Filter the input message...
        smooks.filterSource(executionContext, new
StringSource(messageIn), validationResult);

        return validationResult;
    }
    finally {
        smooks.close();
    }
}
```

CHAPTER 4. PRODUCING OUTPUT DATA

4.1. THE SMOOKS JAVABEAN CARTRIDGE

You can use the Smooks Javabean Cartridge to create and populate Java objects from your message data. It can be used purely as a Java binding framework for XML, EDI, CSV and so forth. However, Smooks' Java binding capabilities are also the cornerstone of many other capabilities. This is because **Smooks** makes the Java objects it creates (and to which it binds data) available through the **`BeanContext`** class. This class is essentially a *Java bean context* that is made available to any Smooks visitor implementation via the Smooks **`ExecutionContext`**.

NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

4.2. JAVABEAN CARTRIDGE FEATURES

- Templating: This usually involves applying a template to the objects in the BeanContext.
- Validation: Business rules validation normally involves applying a rule to the objects in the BeanContext.
- Message splitting and routing: This works by generating split messages from the objects in the BeanContext, either by using the objects themselves and routing them, or by applying a template to them and routing the result of that operation to a new file.
- Persistence: These features depend on the Java binding functions for creating and populating the Java objects (such as entities) that are to be committed to the database. Data read from a database will normally be bound to the BeanContext.
- Message enrichment: Enrichment data (read, for example from a database) is normally bound to the BeanContext, from where it is available to all of Smooks' other features, including the Java binding functionality itself (making it available for *expression-based bindings*.) This allows you to enrich messages generated by Smooks.

4.3. JAVABEAN CARTRIDGE EXAMPLE

The following example is based on this XML:

```
<order>
    <header>
        <date>Wed Nov 15 13:45:28 EST 2006</date>
        <customer number="123123">Joe</customer>
    </header>
    <order-items>
        <order-item>
            <product>111</product>
            <quantity>2</quantity>
            <price>8.90</price>
        </order-item>
        <order-item>
            <product>222</product>
            <quantity>7</quantity>
```

```
            <price>5.20</price>
        </order-item>
    </order-items>
</order>
```

The Javabean Cartridge is used via the http://www.milyn.org/xsd/smooks/javabean-1.4.xsd configuration namespace. (Install the schema in your IDE to avail yourself of the latter's auto-complete functionality.)

Here is an example configuration:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean BeanId="order" class="example.model.Order"
createOnElement="#document" />

</smooks-resource-list>
```

This configuration creates an instance of the **`example.model.Order`** class and binds it to the bean context under the BeanId called **`order`**. The instance is created at the very start of the message on the **`#document`** element (in other words, at the start of the root order element).

The configuration shown above creates the **`example.model.Order`** bean instance and binds it to the bean context.

4.4. JAVABEAN ELEMENTS

- beanId: this is the bean's identifier.
- class: this is the bean's **`fully-qualified class name`**.
- createOnElement: use this attribute to control when the bean instance is to be created. You can control the population of the bean properties through the binding configurations (which are child elements of the **`jb:bean`** element).
- createOnElementNS: you can specify the namespace of the createOnElement via this attribute.

4.5. JAVABEAN CONDITIONS

The Javabean Cartridge sets the following conditions to Java beans:

- There is a public no-argument constructor.
- There are *public property setter methods*. These do not need to follow any specific name formats, but it would be better if they do follow those for the standard property setter method names.
- You cannot set Java bean properties directly.

4.6. JAVABEAN CARTRIDGE DATA BINDINGS

These are the three types of data bindings the Javabean Cartridge allows for:

- jb:value: use this to bind data values from the source message event stream to the target bean.

- jb:wiring: use this to "plug" another bean instance from the bean context into a bean property on the target bean. You can use this configuration to construct an *object graph* (as opposed to a loose collection of Java object instances). You can plug beans in based on their beanId, their Java class type or their annotation.

- jb:expression: use this configuration to bind a value calculated from an expression (in the MVEL language). A simple example is the ability to bind an order item total value to an OrderItem bean (based on the result of an expression that calculates price * quantity). Use the execOnElement attribute expression to define the element on which the expression is to be evaluated and to which the result will be bound. (If you do not define it, the expression is executed based on the value of the parent jb:bean createOnElement.) The value of the targeted element is available in the expression as a string variable under the name **_VALUE** (note the underscore).

4.7. BINDING DATA

1. Using the Order XML message, look at the full XML-to-Java binding configuration. Here are the Java objects that you must populate from that XML message (the "getters" and "setters" are not shown):

```
public class Order {
    private Header header;
    private List<OrderItem> orderItems;
}

public class Header {
    private Date date;
    private Long customerNumber;
    private String customerName;
    private double total;
}

public class OrderItem {
    private long productId;
    private Integer quantity;
    private double price;
}
```

2. Use this configuration to bind the data from the order XML to the object model:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd">

(1)   <jb:bean beanId="order" class="com.acme.Order"
createOnElement="order">
(1.a)     <jb:wiring property="header" beanIdRef="header" />
(1.b)     <jb:wiring property="orderItems" beanIdRef="orderItems" />
      </jb:bean>

(2)   <jb:bean beanId="header" class="com.acme.Header"
createOnElement="order">
(2.a)     <jb:value property="date" decoder="Date"
data="header/date">
              <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
```

```
yyyy</jb:decodeParam>
          </jb:value>
(2.b)     <jb:value property="customerNumber"
data="header/customer/@number" />
(2.c)     <jb:value property="customerName" data="header/customer"
/>
(2.d)     <jb:expression property="total" execOnElement="order-item"
>
              += (orderItem.price * orderItem.quantity);
          </jb:expression>
      </jb:bean>

(3)   <jb:bean beanId="orderItems" class="java.util.ArrayList"
createOnElement="order">
(3.a)     <jb:wiring beanType="com.acme.OrderItem" /> <!-- Could
also wire using beanIdRef="orderItem" -->
      </jb:bean>

(4)   <jb:bean beanId="orderItem" class="com.acme.OrderItem"
createOnElement="order-item">
(4.a)     <jb:value property="productId" data="order-item/product"
/>
(4.b)     <jb:value property="quantity" data="order-item/quantity"
/>
(4.c)     <jb:value property="price" data="order-item/price" />
      </jb:bean>

</smooks-resource-list>
```

4.8. BINDING DATA CONFIGURATIONS

Configuration (1) defines the creation rules for the com.acme.Order bean instance (the top level bean). See the following configurations for details:

- You should create each of the beans instances ((1), (2), (3) but not (4)) at the very start of the message (on the order element). Do this because there will only ever be a single instance of these beans in the populated model.

- Configurations (1.a) and (1.b) define the wiring configuration for wiring the Header and ListOrderItem bean instances ((2) and (3)) into the order bean instance (see the beanIdRef attribute values and how the reference the beanId values defined on (2) and (3)). The property attributes on (1.a) and (1.b) define the Order bean properties on which the wirings are to be made.

  Note also that beans can also be wired into an object based on their Java class type (beanType), or by being annotated with a specific Annotation (beanAnnotation).

  Configuration (2) creates the com.acme.Header bean instance.

- Configuration (2.a) defines a value binding onto the Header.date property. Note that the data attribute defines where the binding value is selected from the source message; in this case it is coming from the header/date element. Also note how it defines a decodeParam sub-element. This configures the DateDecoder.

- Configuration (2.b) defines a value binding configuration onto the Header.customerNumber property. Note how to configure the data attribute to select a binding value from an element attribute on the source message.

  Configuration (2.b) also defines an expression binding where the order total is calculated and set on the Header.total property. The execOnElement attribute tells Smooks that this expression needs to be evaluated (and bound/rebound) on the order-item element. So, if there are multiple order-item elements in the source message, this expression will be executed for each order-item and the new total value rebound into the Header.total property. Note how the expression adds the current orderItem total to the current order total (Header.total).

- Configuration (2.d) defines an expression binding, where a running total is calculated by adding the total for each order item (quantity * price) to the current total. Configuration (3) creates the ListOrderItem bean instance for holding the OrderItem instances.

- Configuration (3.a) wires all beans of type com.acme.OrderItem (i.e. (4)) into the list. Note how this wiring does not define a property attribute. This is because it wires into a Collection (same applies if wiring into an array). You can also perform this wiring using the beanIdRef attribute instead of the beanType attribute.

- Configuration (4) creates the OrderItem bean instances. Note how the createOnElement is set to the order-item element. This allows for a new instance of this bean to be created for every order-item element (and wired into the ListOrderItem (3.a)).

If the createOnElement attribute for this configuration was not set to the order-item element (if, for example, it was set to one of the order, header or order-items elements), then only a single OrderItem bean instance would be created and the binding configurations ((4.a) etc) would overwrite the bean instance property bindings for every order-item element in the source message, that is, you would be left with a ListOrderItem with just a single OrderItem instance containing the order-item data from the last order-item encountered in the source message.

4.9. BINDING TIPS

Here are some binding tips:

- set jb:bean createOnElement to the root element (or **`#document`**) for bean instances where only a single instance will exist in the model.

  Set it to the recurring element for *collection bean instances*.

WARNING

If you do not specify the correct element in this case, you could lose data.

- jb:value decoder: in most cases, Smooks will automatically detect the data-type decoder to be used for a jb:value binding. However, some decoders require configuration (one example being that the DateDecoder [**`decoder="Date"`**]). In these cases, you must define the decoder attribute (and the jb:decodeParam child elements for specifying the decode parameters for that decoder) on the binding.

- jb:wiring property is not required when binding to collections.

- To set the required collection type, define the jb:bean class and wire in the collection entries. For arrays, just *postfix* the jb:bean class attribute value with square brackets (for example, **`class=&"com.acme.OrderItem[]"`**).

4.10. DATADECODER/DATAENCODER IMPLEMENTATIONS

The a DataEncoder implements methods for encoding and formatting an object value to a string. These DataDecoder/DataEncoder implementations are available:

- Date: decodes/encodes a string to a java.util.Date instance.
- Calendar: decodes/encodes a string to a java.util.Calendar instance.
- SqlDate: decodes/encodes a string to a java.sql.Date instance.
- SqlTime: decodes/encodes a string to a java.sql.Time instance.
- SqlTimestamp: decodes/encodes a string to a java.sql.Timestamp instance.

You configure all of these implementations in the same way.

4.11. DATADECODER/DATAENCODER DATE EXAMPLE

Here is a date example:

```
<jb:value property="date" decoder="Date" data="order/@date">
    <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
yyyy</jb:decodeParam>
    <jb:decodeParam name="locale">sv_SE</jb:decodeParam>
</jb:value>
```

4.12. DATADECODER/DATAENCODER SQLTIMESTAMP EXAMPLE

Here is an SqlTimestamp example:

```
<jb:value property="date" decoder="SqlTimestamp" data="order/@date">
    <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
yyyy</jb:decodeParam>
    <jb:decodeParam name="locale">sv</jb:decodeParam>
</jb:value>
```

4.13. DATADECODER/DATAENCODER DECODEPARAM EXAMPLE

The decodeParam format is based on the ISO 8601 standard for date formatting. The locale decodeParam value is an underscore-separated string, with the first token being the ISO language code for the locale and the second token being the ISO country code. This decodeParam can also be specified as two separate parameters for language and country:

```
<jb:value property="date" decoder="Date" data="order/@date">
    <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
yyyy</jb:decodeParam>
```

```
    <jb:decodeParam name="locale-language">sv</jb:decodeParam>
    <jb:decodeParam name="locale-country">SE</jb:decodeParam>
</jb:value>
```

4.14. NUMBER-BASED DATADECODER/DATAENCODER IMPLEMENTATIONS

Several number-based DataDecoder/DataEncoder implementations are available:

- BigDecimalDecoder: use this to decode/encode a string to a java.math. BigDecimal instance.
- BigIntegerDecoder: use this to decode/encode a string to a java.math. BigInteger instance.
- DoubleDecoder: use this to decode/encode a string to a java.lang.Double instance (including primitive).
- FloatDecoder: use this to decode/encode a string to a java.lang.Float instance (including primitive).
- IntegerDecoder: use this to decode/encode a string to a java.lang.Integer instance (including primitive).
- LongDecoder: use this to decode/encode a string to a java.lang.Long instance (including primitive).
- ShortDecoder: use this to decode/encode a string to a java.lang.Short instance (including primitive).

You configure all of these implementations in the same way.

4.15. NUMBER-BASED DATADECODER/DATAENCODER EXAMPLE

Here is a BigDecimal example:

```
<jb:value property="price" decoder="BigDecimal" data="orderItem/price">
    <jb:decodeParam name="format">#,###.##</jb:decodeParam>
    <jb:decodeParam name="locale">en_IE</jb:decodeParam>
</jb:value>
```

4.16. NUMBER-BASED DATADECODER/DATAENCODER INTEGER EXAMPLE

```
<jb:value property="percentage" decoder="Integer" data="vote/percentage">
    <jb:decodeParam name="format">#%</jb:decodeParam>
</jb:value>
```

4.17. NUMBER-BASED DATADECODER/DATAENCODER DECODEPARAM EXAMPLE

The format decodeParam is based on the NumberFormat pattern's syntax. The locale decodeParam value is an underscore-separated string, with the first token being the ISO Language Code for the locale and the second token being the ISO country code. You can also specify this decodeParam as two

separate parameters for language and country. See the example:

```
<jb:value property="price" decoder="Double" data="orderItem/price">
    <jb:decodeParam name="format">#,###.##</jb:decodeParam>
    <jb:decodeParam name="locale-language">sv</jb:decodeParam>
    <jb:decodeParam name="locale-country">SE</jb:decodeParam>
</jb:value>
```

4.18. USING THE MAPPING DECODER TO BIND

1. Configure the Mapping Decoder as shown below to bind a different value to your object model, based on the data in your input message:

```
<jb:value property="name" decoder="Mapping"
data="history/@warehouse">
    <jb:decodeParam name="1">Dublin</jb:decodeParam>
    <jb:decodeParam name="2">Belfast</jb:decodeParam>
    <jb:decodeParam name="3">Cork</jb:decodeParam>
</jb:value>
```

2. An input data value of "1" is mapped to the name property as a value of "Dublin". Likewise for values "2" and "3".

4.19. THE ENUM DECODER

The *Enum Decoder* is a specialized version of the Mapping Decoder. Normally, enumerations are decoded automatically (without any specific configuration needed) if the data input values map exactly to the enum values/names. However when this is not the case, you need to define mappings from the input data value to the enum value/name.

4.20. ENUM DECODER EXAMPLE

1. In the following example, the header/priority field in the input message contains values of **`LOW`**, **`MEDIUM`** and **`HIGH`**. You should map these to the LineOrderPriority enum values of **`NOT_IMPORTANT`**, **`IMPORTANT`**and **`VERY_IMPORTANT`** respectively:

```
<jb:value property="priority" data="header/priority" decoder="Enum">
    <jb:decodeParam
name="enumType">example.trgmodel.LineOrderPriority</jb:decodeParam>
    <jb:decodeParam name="LOW">NOT_IMPORTANT</jb:decodeParam>
    <jb:decodeParam name="MEDIUM">IMPORTANT</jb:decodeParam>
    <jb:decodeParam name="HIGH">VERY_IMPORTANT</jb:decodeParam>
</jb:value>
```

2. If mappings are required, specify the enumeration type using the enumType decodeParam.

4.21. BEANCONTEXT CONFIGURATION

By default, every bean in the Smooks configuration except the first one is removed from the BeanContext after the fragment that created the bean (createOnElement) is processed. (In other words, the bean is added to the BeanContext on the start/visitBefore of the createOnElement fragment, and is

removed from the BeanContext at the end/visitAfter.)

By default, this rule applies to all but the first bean configured in the Smooks configuration. (The first bean is the only bean that is retained in the BeanContext, and so can be accessed after the message has been processed.)

To change this behaviour, use the jb:bean element's retain configuration attribute. This attribute allows you to manually control bean retention within the Smooks BeanContext.

4.22. JAVABEAN CARTRIDGE ACTIONS

The Java Bean cartridge:

- extracts string values from the source/input message stream.
- decodes the string value based on the **`decoder`** and **`decodeParam`** configurations (if these are not defined, an attempt is made to reflectively resolve the decoder).
- sets the decoded value on the target bean.

4.23. PRE-PROCESSING STRING DATA

Before decoding, you may need to *pre-process* the string data value. An example of this is when the source data has some characters not supported by the locale configuration on Numeric Decoding, such as the numeric value **`876592.00`** being represented as **`876_592!00`**. To decode this value as (for instance) a double value, delete the underscore and replace the exclamation mark with a full-stop. You can specify a valuePreprocess decodeParam, which is a simple expression that you can applied to the String value before decoding it.

4.24. PRE-PROCESSING EXAMPLE

This example provides a solution to a numeric decoding issue:

```
<!-- A bean property binding example: -->
<jb:bean beanId="orderItem" class="org.milyn.javabean.OrderItem"
createOnElement="price">
    <jb:value property="price" data="price" decoder="Double">
        <jb:decodeParam name="valuePreprocess">value.replace("_",
"").replace("!", ".")</jb:decodeParam>
    </jb:value>
</jb:bean>
```

```
<!-- A direct value binding example: -->
<jb:value beanId="price" data="price" decoder="BigDecimal">
    <jb:decodeParam name="valuePreprocess">value.replace("_",
"").replace("!", ".")</jb:decodeParam>
</jb:value>
```

The String data value is referenced in the expression via the value variable name. (The expression can be any valid MVEL expression that operates on the value String and returns a String.)

4.25. THE JAVABEAN CARTRIDGE AND FACTORIES

The Javabean Cartridge allows you to use *factories* to create beans. In these cases you do not need to use a *public parameterless constructor*. You do not need to have defined the actual class name in the class attribute. Any of the object's interfaces suffice. However you can only bind to that interface's methods. (Even if you define a factory, you must always set the class attribute in the bean definition.)

The factory definition is set in the bean element's factory attribute. The default factory definition language looks like this:

```
some.package.FactoryClass#staticMethod{.instanceMethod}
```

Use this basic definition language to define a static public parameterless method that Smooks will call to create the bean. (The instanceMethod part is optional. If you set it, it defines the method that will be called on the object that is returned from static method, which should create the bean. The { } characters are only there to illustrate that the part is optional and should be left out of the actual definition.)

4.26. INSTANTIATE AN ARRAYLIST OBJECT USING A STATIC FACTORY METHOD

1. Follow this example to instantiate an ArrayList object using a static factory method:

```
<jb:bean
   beanId="orders"
   class="java.util.List"
   factory="some.package.ListFactory#newList"
   createOnElement="orders"
>
     <!-- ... bindings -->
</jb:bean>
```

   The **`some.package.ListFactory#newList`** factory definition establishes that the **`newList`** method must be called on the **`some.package.ListFactory`** class in order to create the bean. The class attributes define the bean as a List object. The specific kind of List object that it is (be it an ArrayList or a LinkedList), is decided by the ListFactory itself.

2. Observe this additional example:

```
<jb:bean
   beanId="orders"
   class="java.util.List"
   factory="some.package.ListFactory#getInstance.newList"
   createOnElement="orders"
>
     <!-- ... bindings -->
</jb:bean>
```

   This defines that an instance of the ListFactory needs to be retrieved using the static method **`getInstance`** and then the **`newList`** method needs to be called on the ListFactory object to create the List object. This construct lets you use *singleton factories.*

4.27. DECLARING DEFINITION LANGUAGE

These are the ways you can declare which definition language you want to use:

- Each definition language can have an alias. For instance MVEL has the alias mvel. To define that you want to use MVEL for a specific factory definition you put mvel: in front of the definition. For example. mvel:some.package.ListFactory.getInstance().newList(). The alias of the default basic language is basic.

- To set a language as a global default you need to set the 'factory.definition.parser.class' global parameter to the full class path of the class that implements the FactoryDefinitionParser interface for the language that you want to use. If you have a definition with your default language that includes a `:` you should prefix that definition with **`default:`** to avoid an exception.

- You can set the full classpath of the class that implements the FactoryDefinitionParser interface for the language that you want to use. (For example, org.milyn.javabean.factory.MVELFactoryDefinitionParser:some.package.ListFactory.getInstance().r You should generally use this for test purposes only. It is much better to define an alias for your language.

4.28. USING YOUR OWN DEFINITION LANGUAGE

1. To define your own language, implement the org.milyn.javabean.factory.FactoryDefinitionParser interface. Observe the org.milyn.javabean.factory.MVELFactoryDefinitionParser or org.milyn.javabean.factory.BasicFactoryDefinitionParser for examples.

2. To define the alias for a definition language, add the org.milyn.javabean.factory.Alias annotation with the alias name to your FactoryDefinitionParser class.

3. For Smooks to find your alias you need create the file META-INF/smooks-javabean-factory-definition-parsers.inf on the root of your classpath. T

4.29. THE MVEL LANGUAGE

MVEL has some advantages over the basic default definition language. It lets you use objects from the bean context as the factory object and allows you to call factory methods with parameters. These parameters can be defined within the definition or they can be objects from the bean context. To use MVEL, use the alias **`mvel`** or set the **`factory.definition.parser.class`** global parameter to **`org.milyn.javabean.factory.MVELFactoryDefinitionParser`**.

4.30. MVEL EXAMPLE

```
<smooks-resource-list
   xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
   xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean
        beanId="orders"
        class="java.util.List"
        factory="mvel:some.package.ListFactory.getInstance().newList()"
        createOnElement="orders"
    >
        <!-- ... bindings -->
    </jb:bean>

</smooks-resource-list>
```

4.31. EXTRACTING A LIST OBJECT WITH MVEL

To use MVEL to extract a List object from an existing bean in the bean context, see the example below. (The Order object in this example has a method that returns a list which we must use to add order lines.)

```
<smooks-resource-list
   xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
   xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean
        beanId="order"
        class="some.package.Order"
        createOnElement="order"
    >
        <!-- ... bindings -->
    </jb:bean>

    <!--
           The factory attribute uses MVEL to access the order
           object in the bean context and calls its getOrderLines()
           method to get the List. This list is then added to the bean
           context under the beanId 'orderLines'
    -->
    <jb:bean
        BeanId="orderLines"
        class="java.util.List"
        factory="mvel:order.getOrderLines()"
        createOnElement="order"
    >
        <jb:wiring BeanIdRef="orderLine" />
    </jb:bean>

    <jb:bean
        BeanId="orderLine"
        class="java.util.List"
        createOnElement="order-line"
    >
        <!-- ... bindings -->
    </jb:bean>

</smooks-resource-list>
```

NOTE

Array objects are not supported. If a factory returns an array then Smooks will throw an exception.

4.32. THE JB:VALUE PROPERTY

If the jb:value property attribute of a binding is not defined when binding key value pairs into maps, the name of the selected node will be used as the map entry key (where the beanClass is a Map). You can also define the jb:value property attribute by putting the "@" character in front of it. The rest of the value then defines the attribute name of the selected node, from which the map key is selected.

4.33. JB:VALUE PROPERTY EXAMPLE

```
<root>
    <property name="key1">value1</property>
    <property name="key2">value2</property>
    <property name="key3">value3</property>
</root>
```

On the config:

```
<jb:bean BeanId="keyValuePairs" class="java.util.HashMap"
createOnElement="root">
    <jb:value property="@name" data="root/property" />
</jb:bean>
```

This would create a HashMap with three entries with the keys set [key1, key2, key3].

NOTE

The "@" character notation doesn't work for bean wiring. The cartridge will simply use the value of the property attribute, including the "@" character, as the map entry key.

4.34. VIRTUAL OBJECT MODELS

You can create a *virtual object model* without writing your own Bean classes. This virtual model is created using only maps and lists. This is convenient if you use the Javabean Cartridge between two processing steps. For example, as part of a model driven transform - xml to java to xml or xml to java to edi.

4.35. VIRTUAL OBJECT MODEL EXAMPLE

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    Bind data from the message into a Virtual Object model in the bean
context....
    -->
    <jb:bean beanId="order" class="java.util.HashMap"
createOnElement="order">
        <jb:wiring property="header" beanIdRef="header" />
        <jb:wiring property="orderItems" beanIdRef="orderItems" />
    </jb:bean>
    <jb:bean beanId="header" class="java.util.HashMap"
createOnElement="order">
        <jb:value property="date" decoder="Date" data="header/date">
            <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
yyyy</jb:decodeParam>
        </jb:value>
```

```
        <jb:value property="customerNumber" decoder="Long"
data="header/customer/@number" />
        <jb:value property="customerName" data="header/customer" />
        <jb:expression property="total" execOnElement="order-item" >
            header.total + (orderItem.price * orderItem.quantity);
        </jb:expression>
    </jb:bean>
    <jb:bean beanId="orderItems" class="java.util.ArrayList"
createOnElement="order">
        <jb:wiring beanIdRef="orderItem" />
    </jb:bean>
    <jb:bean beanId="orderItem" class="java.util.HashMap"
createOnElement="order-item">
        <jb:value property="productId" decoder="Long" data="order-
item/product" />
        <jb:value property="quantity" decoder="Integer" data="order-
item/quantity" />
        <jb:value property="price" decoder="Double" data="order-
item/price" />
    </jb:bean>

    <!--
    Use a FreeMarker template to perform the model driven transformation
on the Virtual Object Model...
    -->
    <ftl:freemarker applyOnElement="order">
        <ftl:template>/templates/orderA-to-orderB.ftl</ftl:template>
    </ftl:freemarker>

</smooks-resource-list>
```

Note above how the decoder attribute for a Virtual Model (Map) is always defined. This is because Smooks has no way of auto-detecting the decode type for data binding to a Map. If you need typed values bound into your Virtual Model, you need to specify an appropriate decoder. If the decoder is not specified in this case, Smooks will simply bind the data into the Virtual Model as a String.

4.36. MERGING MULTIPLE DATA ENTITIES INTO A SINGLE BINDING

You can merge multiple data entities into a single binding using Expression Based Bindings (**`jb:expression`**). The Javabean cartridge uses the Smooks DataDecoder to create an Object from a selected data element/attribute. It then adds it directly to the bean context.

4.37. VALUE BINDING

The ValueBinder class is the visitor that does the value binding. The value binding XML configuration is part of the JavaBean schema from Smooks 1.3 on http://www.milyn.org/xsd/smooks/javabean-1.4.xsd. The element for the value binding is "value".

4.38. VALUE BINDING ATTRIBUTES

The ValueBinder class has the following attributes:

- beanId: The ID under which the created object is to be bound in the bean context.

- data: The data selector for the data value to be bound. (For example, **`order/orderid`** or **`order/header/@date`**
- dataNS: The namespace for the data selector
- decoder: The DataDecoder name for converting the value from a String into a different type. The DataDecoder can be configured with the decodeParam elements.
- default: The default value for if the selected data is null or an empty string.

4.39. VALUE BINDING EXAMPLE

The Order message will be used as an example. It will be configured for getting the order number, name and date as Value Objects in the form of an Integer and String.

The message input:

```
<order xmlns="http://x">
     <header>
         <y:date xmlns:y="http://y">Wed Nov 15 13:45:28 EST 2006</y:date>
         <customer number="123123">Joe</customer>
         <privatePerson></privatePerson>
     </header>
     <order-items>
         <!-- .... -->
     </order-items>
 </order>
```

The configuration:

```
<?xml version="1.0"?>
 <smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:value
       beanId="customerName"
       data="customer"
       default="unknown"
    />

    <jb:value
       beanId="customerNumber"
       data="customer/@number"
       decoder="Integer"
    />

   <jb:value
       beanId="orderDate"
       data="date"
       dateNS="http://y"
       decoder="Date"
    >
         <jb:decodeParam name="format">EEE MMM dd HH:mm:ss z
yyyy</jb:decodeParam>
         <jb:decodeParam name="locale-language">en</jb:decodeParam>
```

```
        <jb:decodeParam name="locale-country">IE</jb:decodeParam>
    </jb:value>

 </smooks-resource-list>
```

4.40. PROGRAMMATIC VALUE BINDING EXAMPLE

The value binder can be programmatically configured using the **`org.milyn.javabean.Value`** object:

```
// Create Smooks. normally done globally!
Smooks smooks = new Smooks();

// Create the Value visitors
Value customerNumberValue = new Value( "customerNumber",
"customer/@number").setDecoder("Integer");
Value customerNameValue = new Value( "customerName",
"customer").setDefault("Unknown");

// Add the Value visitors
smooks.addVisitors(customerNumberValue);
smooks.addVisitors(customerNameValue);

// And the execution code:
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(orderMessageStream), result);
Integer customerNumber = (Integer) result.getBean("customerNumber");
String customerName = (String) result.getBean("customerName");
```

4.41. JAVA BINDING IN SMOOKS

Java Binding Configurations can be programmatically added to a Smooks using the Bean configuration class. This class can be used to programmatically configure a Smooks instance for performing a Java Bindings on a specific class. To populate a graph, you simply create a graph of bean instances by binding beans onto beans. The bean class uses a Fluent API (all methods return the bean instance), making it easy to string configurations together to build up a graph of bean configuration.

4.42. JAVA BINDING EXAMPLE

This Order message example shows how to bind it into a corresponding Java Object model.

The message input:

```
<order xmlns="http://x">
    <header>
        <y:date xmlns:y="http://y">Wed Nov 15 13:45:28 EST 2006</y:date>
        <customer number="123123">Joe</customer>
        <privatePerson></privatePerson>
    </header>
    <order-items>
        <order-item>
            <product>111</product>
            <quantity>2</quantity>
```

```
            <price>8.90</price>
        </order-item>
        <order-item>
            <product>222</product>
            <quantity>7</quantity>
            <price>5.20</price>
        </order-item>
    </order-items>
</order>
```

The Java Model (not including getters/setters):

```
public class Order {
    private Header header;
    private List<OrderItem> orderItems;
}

public class Header {
    private Long customerNumber;
    private String customerName;
}

public class OrderItem {
    private long productId;
    private Integer quantity;
    private double price;
}
```

The configuration code:

```
Smooks smooks = new Smooks();

Bean orderBean = new Bean(Order.class, "order", "/order");

orderBean.bindTo("header",
    orderBean.newBean(Header.class, "/order")
        .bindTo("customerNumber", "header/customer/@number")
        .bindTo("customerName", "header/customer")
    ).bindTo("orderItems",
    orderBean.newBean(ArrayList.class, "/order")
        .bindTo(orderBean.newBean(OrderItem.class, "order-item")
            .bindTo("productId", "order-item/product")
            .bindTo("quantity", "order-item/quantity")
            .bindTo("price", "order-item/price"))
    );

smooks.addVisitors(orderBean);
```

The execution code:

```
 JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(orderMessageStream), result);
Order order = (Order) result.getBean("order");
```

Here is an example where an anonymous Factory class is defined and used:

```
Bean orderBean = new Bean(Order.class, "order", "/order", new
Factory<Order>() {

 public Order create(ExecutionContext executionContext) {
  return new Order();
 }

});
```

4.43. THE ORG.MILYN.JAVABEAN.GEN.CONFIGGENERATOR UTILITY CLASS

The Javabean Cartridge contains the org.milyn.javabean.gen.ConfigGenerator utility class that can be used to generate a binding configuration template. This template can then be used as the basis for defining a binding.

4.44. ORG.MILYN.JAVABEAN.GEN.CONFIGGENERATOR EXAMPLE

From the commandline:

```
$JAVA_HOME/bin/java -classpath <classpath>
org.milyn.javabean.gen.ConfigGenerator -c <rootBeanClass> -o
<outputFilePath> [-p <propertiesFilePath>]
```

- The **-c** commandline arg specifies the root class of the model whose binding config is to be generated.
- The **-o** commandline arg specifies the path and filename for the generated config output.
- The **-p** commandline arg specifies the path and filename optional binding configuration file that specifies additional binding parameters.
- The optional **-p** properties file parameter allows specification of additional config parameters.
- packages.included: Semi-colon separated list of packages. Any fields in the class matching these packages will be included in the binding configuration generated.
- packages.excluded: Semi-colon separated list of packages. Any fields in the class matching these packages will be excluded from the binding configuration generated.

4.45. PROGRAMMING THE BINDING CONFIGURATION

After running the the org.milyn.javabean.gen.ConfigGenerator utility class against the target class, you should perform these tasks to make the binding configuration work for your Source data model.

1. For each jb:bean element, set the createOnElement attribute to the event element that should be used to create the bean instance.
2. Update the jb:value data attributes to select the event element/attribute supplying the binding data for that BFean property.

3. Check the jb:value decoder attributes. Not all will be set, depending on the actual property type. These must be configured by hand. You may need to configure jb:decodeParam sub-elements for the decoder on some of the bindings, for example, on a date field.

4. Double-check the binding configuration elements (jb:value and jb:wiring), making sure all Java properties have been covered in the generated configuration.

4.46. CONFIGURING TRANSFORMATIONS

1. Access the HTML Reporting Tool when determining selector values. It helps you visualise the input message model (against which the selectors will be applied) as seen by Smooks.

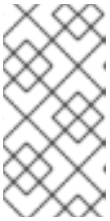
2. Generate a report using your Source data, but with an empty transformation configuration. In the report, you can see the model against which you need to add your configurations. Add the configurations one at a time, rerunning the report to check they are being applied.

3. Add the configurations one at a time, rerunning the report to check they are being applied.

4. As a result, a configuration that looks like this will be generated (note the **$TODO$** tokens):

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd">

    <jb:bean beanId="order" class="org.milyn.javabean.Order"
createOnElement="$TODO$">
        <jb:wiring property="header" beanIdRef="header" />
        <jb:wiring property="orderItems" beanIdRef="orderItems" />
        <jb:wiring property="orderItemsArray"
beanIdRef="orderItemsArray" />
    </jb:bean>

    <jb:bean beanId="header" class="org.milyn.javabean.Header"
createOnElement="$TODO$">
        <jb:value property="date" decoder="$TODO$" data="$TODO$" />
        <jb:value property="customerNumber" decoder="Long"
data="$TODO$" />
        <jb:value property="customerName" decoder="String"
data="$TODO$" />
        <jb:value property="privatePerson" decoder="Boolean"
data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

    <jb:bean beanId="orderItems" class="java.util.ArrayList"
createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItems_entry" />
    </jb:bean>

    <jb:bean beanId="orderItems_entry"
class="org.milyn.javabean.OrderItem" createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$"
/>
        <jb:value property="quantity" decoder="Integer"
data="$TODO$" />
```

```
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

    <jb:bean beanId="orderItemsArray"
class="org.milyn.javabean.OrderItem[]" createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItemsArray_entry" />
    </jb:bean>

    <jb:bean beanId="orderItemsArray_entry"
class="org.milyn.javabean.OrderItem" createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$"
/>
        <jb:value property="quantity" decoder="Integer"
data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

</smooks-resource-list>
```



NOTE

There is no guarantee as to the exact contents of a JavaResult instance after calling the Smooks.filterSource method. After calling this method, the JavaResult instance will contain the final contents of the bean context, which can be added to by any Visitor implementation.

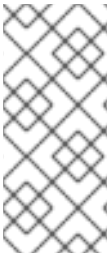
4.47. JB:RESULT CONFIGURATION EXAMPLE

You can restrict the Bean set returned in a JavaResult by using a *jb:result* configuration in the Smooks configuration. In the following example configuration, we tell Smooks to only retain the **`order`** bean in the ResultSet:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd">

    <!-- Capture some data from the message into the bean context... -->
    <jb:bean beanId="order" class="com.acme.Order"
createOnElement="order">
        <jb:value property="orderId" data="order/@id"/>
        <jb:value property="customerNumber"
data="header/customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
        <jb:wiring property="orderItems" beanIdRef="orderItems"/>
    </jb:bean>
    <jb:bean beanId="orderItems" class="java.util.ArrayList"
createOnElement="order">
        <jb:wiring beanIdRef="orderItem"/>
    </jb:bean>
    <jb:bean beanId="orderItem" class="com.acme.OrderItem"
createOnElement="order-item">
        <jb:value property="itemId" data="order-item/@id"/>
```

```
        <jb:value property="productId" data="order-item/product"/>
        <jb:value property="quantity" data="order-item/quantity"/>
        <jb:value property="price" data="order-item/price"/>
    </jb:bean>

    <!-- Only retain the "order" bean in the root of any final JavaResult.
-->
    <jb:result retainBeans="order"/>

</smooks-resource-list>
```

After applying this configuration, any calls to the JavaResult.getBean(String) method for non-**`order`** Bean results will return null. This will work in cases such as the above example, because the other bean instances are wired into the **`order`** graph.



NOTE

Note that as of Smooks v1.2, if a JavaSource instance is supplied to the Smooks.filterSource method (as the filter Source instance), Smooks will use the JavaSource to construct the bean context associated with the ExecutionContect for that Smooks.filterSource invocation. This will mean that some of the JavaSource bean instances may be visible in the JavaResult.

CHAPTER 5. TEMPLATES

5.1. SMOOKS TEMPLATES

NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

The two kinds of templates available in Smooks are FreeMarker (http://freemarker.org) and XSL (http://www.w3.org/Style/XSL/).

These technologies can be used within the context of a Smooks filtering process. This means that they:

- can be applied to the source message on a *per-fragment* basis. This is in contrast to the fragment-based transformation process which is applied the whole message. Applying them on a per-fragment basis is useful when there is a need to insert a piece of data into a message at a very specific point, such as when adding a header to a SOAP message. In this case, the process can be "aimed" at the fragment of interest without disrupting the rest of it.
- can take advantage of other Smooks technologies such as the Javabean Cartridge, which can be used to decode and bind message data to the bean context. It can then make reference to that decoded data from within the **FreeMarker** template. (Smooks makes data available to **FreeMarker**.)
- can be used to process huge; message streams (those which are many gigabytes in size) while at the same time maintaining a simple processing model and a small memory footprint.
- can be used to generate *split message fragments*. These can then be routed to physical or logical endpoints on an enterprise service bus.

5.2. FREEMARKER TEMPLATES

FreeMarker is a very powerful *template engine*. Smooks can use **FreeMarker** as a means of generating text-based content. This content can then be inserted into a message stream (this process is known as a *fragment-based transformation*). The process can also be used to split message fragments for subsequent routing to another process.

Set Smooks' **FreeMarker** templates by using the configuration namespace http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd. Then, configure the XSD in an integrated development environment in order to begin using it.

5.3. FREEMARKER TEMPLATE EXAMPLES

- Inline template example:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-
1.1.xsd">
    <ftl:freemarker applyOnElement="order">
        <ftl:template><!--<orderId>${order.id}</orderId>-->
```

```
</ftl:template>
    </ftl:freemarker>
</smooks-resource-list>
```

- FreeMarker external template example:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-
1.1.xsd">
    <ftl:freemarker applyOnElement="order">
        <ftl:template>/templates/shop/ordergen.ftl</ftl:template>
    </ftl:freemarker>
</smooks-resource-list>
```

- Add the <use> element to the <ftl:freemarker> configuration in order to allow Smooks to perform a number of operations upon the resulting output. See the example below:

```
<ftl:freemarker applyOnElement="order">
    <ftl:template>/templates/shop/ordergen.ftl</ftl:template>
    <ftl:use>
        <ftl:inline directive="insertbefore" />
    </ftl:use>
</ftl:freemarker>
```

5.4. INLINING IN SMOOKS

Inlining, as its name implies, allows you to "inline" the templating result to a **`Smooks.filterSource`** result. A number of directives are supported:

- addto: this adds the templating result to the targeted element.
- replace (default): this uses the templating result to replace the targeted element. This is the default behavior for the <ftl:freemarker> configuration when the <use> element is not configured.
- insertbefore: this adds the templating result before the targeted element.
- insertafter: this adds the templating result after the targeted element.

5.5. THE FTL:BINDTO ELEMENT

The *ftl:bindTo* element allows you to bind a templating result to the Smooks bean context. The result can then be accessed by other Smooks components, such as those used for routing. This is especially useful when you are splitting huge; messages into smaller ones. The split fragments can then be routed to another process.

5.6. FTL:BINDTO EXAMPLE

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jms="http://www.milyn.org/xsd/smooks/jms-
routing-1.2.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">
```

```
    <jms:router routeOnElement="order-item" beanId="orderItem_xml"
destination="queue.orderItems" />

    <ftl:freemarker applyOnElement="order-item">
        <ftl:template>/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!-- Bind the templating result into the bean context, from
where
                 it can be accessed by the JMSRouter (configured above).
-->
            <ftl:bindTo id="orderItem_xml"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>
```

5.7. THE FTL:OUTPUTTO ELEMENT

The *ftl:outputTo* can be used to write an output result directly to an **`OutputStreamResource`** class. This is another useful mechanism for splitting huge messages into smaller ones.

5.8. FTL:OUTPUTTO EXAMPLE

Example of writing the template result to an **`OutputStreamSource`**:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.3.xsd"
                      xmlns:file="http://www.milyn.org/xsd/smooks/file-
routing-1.1.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!-- Create/open a file output stream. This is written to by the
freemarker template (below).. -->
    <file:outputStream openOnElement="order-item"
resourceName="orderItemSplitStream">
        <file:fileNamePattern>order-${order.orderId}-
${order.orderItem.itemId}.xml</file:fileNamePattern>

<file:destinationDirectoryPattern>target/orders</file:destinationDirectory
Pattern>
        <file:listFileNamePattern>order-
${order.orderId}.lst</file:listFileNamePattern>

        <file:highWaterMark mark="3"/>
    </file:outputStream>

    <!--
    Every time we hit the end of an <order-item> element, apply this
freemarker template,
    outputting the result to the "orderItemSplitStream" OutputStream,
which is the file
    output stream configured above.
```

```
    -->
     <ftl:freemarker applyOnElement="order-item">
        <ftl:template>target/classes/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!-- Output the templating result to the
"orderItemSplitStream" file output stream... -->
            <ftl:outputTo outputStreamResource="orderItemSplitStream"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>
```

NOTE

A comprehensive tutorial can be found at http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.3_Examples.

5.9. CONFIGURING TRANSFORMATIONS

1. Access the HTML Reporting Tool when determining selector values. It helps you visualise the input message model (against which the selectors will be applied) as seen by Smooks.
2. Generate a report using your Source data, but with an empty transformation configuration. In the report, you can see the model against which you need to add your configurations. Add the configurations one at a time, rerunning the report to check they are being applied.
3. Add the configurations one at a time, rerunning the report to check they are being applied.
4. As a result, a configuration that looks like this will be generated (note the **$TODO$** tokens):

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd">

    <jb:bean beanId="order" class="org.milyn.javabean.Order"
createOnElement="$TODO$">
        <jb:wiring property="header" beanIdRef="header" />
        <jb:wiring property="orderItems" beanIdRef="orderItems" />
        <jb:wiring property="orderItemsArray"
beanIdRef="orderItemsArray" />
    </jb:bean>

    <jb:bean beanId="header" class="org.milyn.javabean.Header"
createOnElement="$TODO$">
        <jb:value property="date" decoder="$TODO$" data="$TODO$" />
        <jb:value property="customerNumber" decoder="Long"
data="$TODO$" />
        <jb:value property="customerName" decoder="String"
data="$TODO$" />
        <jb:value property="privatePerson" decoder="Boolean"
data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>
```

```
    <jb:bean beanId="orderItems" class="java.util.ArrayList"
createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItems_entry" />
    </jb:bean>

    <jb:bean beanId="orderItems_entry"
class="org.milyn.javabean.OrderItem" createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$"
/>
        <jb:value property="quantity" decoder="Integer"
data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

    <jb:bean beanId="orderItemsArray"
class="org.milyn.javabean.OrderItem[]" createOnElement="$TODO$">
        <jb:wiring beanIdRef="orderItemsArray_entry" />
    </jb:bean>

    <jb:bean beanId="orderItemsArray_entry"
class="org.milyn.javabean.OrderItem" createOnElement="$TODO$">
        <jb:value property="productId" decoder="Long" data="$TODO$"
/>
        <jb:value property="quantity" decoder="Integer"
data="$TODO$" />
        <jb:value property="price" decoder="Double" data="$TODO$" />
        <jb:wiring property="order" beanIdRef="order" />
    </jb:bean>

</smooks-resource-list>
```

NOTE

There is no guarantee as to the exact contents of a JavaResult instance after calling the Smooks.filterSource method. After calling this method, the JavaResult instance will contain the final contents of the bean context, which can be added to by any Visitor implementation.

5.10. FREEMARKER AND THE JAVA BEAN CARTRIDGE

The **`FreeMarkerNodeModel`** is powerful and easy to use, but this comes at a trade-off in terms of performance. It is not "cheap" to construct W3C DOMs. It also may be the case that the required data has already been extracted and populated into a Java object model, an example being when the data also needs to be routed to a Java Message Service endpoint as a set of objects.

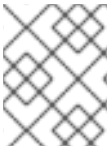
When using the **`NodeModel`** would be impractical, use the Java Bean Cartridge to populate a proper Java object or a virtual model. This model can then be used in the **FreeMarker** templating process.

5.11. NODEMODEL EXAMPLE

The following example shows you how to configure the NodeModel element:

```
<?xml version="1.0"?>
```

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.3.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!-- Extract and decode data from the message. Used in the freemarker
template (below). -->
    <jb:bean beanId="order" class="java.util.Hashtable"
createOnElement="order">
        <jb:value property="orderId" decoder="Integer" data="order/@id"/>
        <jb:value property="customerNumber" decoder="Long"
data="header/customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
        <jb:wiring property="orderItem" beanIdRef="orderItem"/>
    </jb:bean>
    <jb:bean beanId="orderItem" class="java.util.Hashtable"
createOnElement="order-item">
        <jb:value property="itemId" decoder="Integer" data="order-
item/@id"/>
        <jb:value property="productId" decoder="Long" data="order-
item/product"/>
        <jb:value property="quantity" decoder="Integer" data="order-
item/quantity"/>
        <jb:value property="price" decoder="Double" data="order-
item/price"/>
    </jb:bean>

    <ftl:freemarker applyOnElement="order-item">
        <ftl:template><!--<orderitem id="${order.orderItem.itemId}"
order="${order.orderId}">
    <customer>
        <name>${order.customerName}</name>
        <number>${order.customerNumber?c}</number>
    </customer>
    <details>
        <productId>${order.orderItem.productId}</productId>
        <quantity>${order.orderItem.quantity}</quantity>
        <price>${order.orderItem.price}</price>
    </details>
</orderitem>-->
        </ftl:template>
    </ftl:freemarker>

</smooks-resource-list>
```



NOTE

An extended example can be seen at http://www.smooks.org/mediawiki/index.php?title=Smooks_v1.3_Examples.

5.12. PROGRAMMATIC CONFIGURATION

FreeMarker template configurations can be added to a Smooks instance programmatically. Do so by adding and configuring a **`FreeMarkerTemplateProcessor`** instance as shown below. This example adds configurations for a Java binding and a **FreeMarker** template to Smooks:

```
Smooks smooks = new Smooks();

smooks.addVisitor(new Bean(OrderItem.class, "orderItem", "order-
item").bindTo("productId", "order-item/product/@id"));
smooks.addVisitor(new FreeMarkerTemplateProcessor(new
TemplatingConfiguration("/templates/order-tem.ftl")), "order-item");

// Then use Smooks as normal... filter a Source to a Result etc...
```

5.13. XSL TEMPLATES

- To use XSL templates in Smooks, configure the http://www.milyn.org/xsd/smooks/xsl-1.1.xsd XSD in an integrated development environment.

IMPORTANT

In Red Hat JBoss Fuse, the fragment filter is bypassed when the Smooks configuration only contains a single XSLT that is applied to the root fragment. The XSLT is applied directly. This is done for performance reasons and can be disabled by adding a parameter called **`enableFilterBypass`** and setting it to **`false`**:

```
<param name="enableFilterBypass">false</param>
```

5.14. XSL EXAMPLE

The following example shows you how to configure an XSL template:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:xsl="http://www.milyn.org/xsd/smooks/xsl-1.1.xsd">

    <xsl:xsl applyOnElement="#document">
        <xsl:template><!--<xxxxxx/>--></xsl:template>
    </xsl:xsl>

</smooks-resource-list>
```

As is the case with **FreeMarker**, other types of external template can be configured using an URI reference in the <xsl:template> element.

5.15. POINTS TO NOTE REGARDING XSL SUPPORT

There is no reason to use Smooks to execute XSL templates unless:

- there is a need to perform a fragment transformation, as opposed to the transformation of a whole message.

- there is a need to use other Smooks functionality to perform additional operations on the message, such as splitting or persistence.

- XSL templating is only supported through the DOM filter. It is not supported through the SAX filter. This can (depending on the XSL being applied) result in lower performance when compared to a SAX-based application of XSL.

- Smooks applies XSL templates on a per-message fragment basis. This can be very useful for fragmenting XSLs, but do not assume that a template written for a stand-alone context will automatically work in Smooks without modification. For this reason, Smooks handles XSLs targeted at the document root node differently in that it applies the template to the DOM Document Node (rather than the root DOM Element.)

- most XSLs contain a template that is matched to the root element. Because Smooks applies XSLs on a per-fragment basis, this is no longer valid. Ensure that the style-sheet contains a template that matches against the context node instead (that is, the targeted fragment).

5.16. POTENTIAL ISSUE: XSLT WORKS EXTERNALLY BUT NOT WITHIN SMOOKS

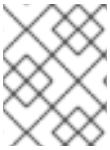
This can happen on occasions and normally results from one of the following scenarios:

- Issues will occur in the Smooks Fragment-Based Processing Model if the stylesheet contains a template that is using an absolute path reference to the document root node. This is because the wrong element is being targeted by Smooks. To rectify the problem, ensure that the XSLT contains a template that matches the context node being targeted by Smooks.

- *SAX versus DOM Processing*: "like" is not being compared with "like". In its current state, Smooks only supports a DOM-based processing model for dealing with XSL. In order to undertake an accurate comparison, use a *DOMSource* (one that is namespace-aware) when executing the XSL template outside Smooks. (A given XSL Processor does not always produce the same output when it tries to apply an XSL template using SAX or DOM.)

CHAPTER 6. ENRICHING OUTPUT DATA

6.1. OUT-OF-THE-BOX ENRICHMENT METHODS



NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

Three methods for enriching your output data are included with the product:

JDBC Datasources

Use a JDBC Datasource to access a database and use SQL statements to read from and write to the Database. This capability is provided through the Smooks Routing Cartridge. See the section on routing to a database using SQL.

Entity persistence

Use an entity persistence framework (like Ibatis, Hibernate or any JPA compatible framework) to access a database and use its query language or CRUD methods to read from it or write to it.

DAOs

Use custom Data Access Objects (DAOs) to access a database and use its CRUD methods to read from it or write to it.

6.2. HIBERNATION EXAMPLE

The data to be processed is an XML order message. Depending on your needs, the input data could also be CSV, JSON, EDI, Java or any other structured/hierarchical data format. The same principals apply, no matter what the data format is. The same principals follow for any JPA compliant framework:

```
<order>
    <ordernumber>1</ordernumber>
    <customer>123456</customer>
    <order-items>
        <order-item>
            <product>11</product>
            <quantity>2</quantity>
        </order-item>
        <order-item>
            <product>22</product>
            <quantity>7</quantity>
        </order-item>
    </order-items>
</order>
```

6.3. HIBERNATE ENTITIES

The following is a snapshot of the hibernate function's entities:

```
@Entity
```

```
@Table(name="orders")
public class Order {

    @Id
    private Integer ordernumber;

    @Basic
    private String customerId;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL)
    private List orderItems = new ArrayList();

    public void addOrderLine(OrderLine orderLine) {
        orderItems.add(orderLine);
    }

    // Getters and Setters....
}

@Entity
@Table(name="orderlines")
public class OrderLine {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne
    @JoinColumn(name="orderid")
    private Order order;

    @Basic
    private Integer quantity;

    @ManyToOne
    @JoinColumn(name="productid")
    private Product product;

    // Getters and Setters....
}

@Entity
@Table(name = "products")
@NamedQuery(name="product.byId", query="from Product p where p.id = :id")
public class Product {

    @Id
    private Integer id;

    @Basic
    private String name;

    // Getters and Setters....
}
```

6.4. PROCESSING AND PERSISTING AN ORDER

1. To process and persist an XML "order" message, you should bind the order data into the Order entities (Order, OrderLine and Product). To do this, create and populate the Order and OrderLine entities using the Java Binding framework.

2. Wire each OrderLine instance into the Order instance.

3. In each OrderLine instance, you should lookup and wire in the associated order line Product entity.

4. Finally, insert (persist) the Order instance as seen below:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"

xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd"

xmlns:dao="http://www.milyn.org/xsd/smooks/persistence-1.2.xsd">

    <jb:bean beanId="order" class="example.entity.Order"
createOnElement="order">
        <jb:value property="ordernumber" data="ordernumber" />
        <jb:value property="customerId" data="customer" />
        <jb:wiring setterMethod="addOrderLine" beanIdRef="orderLine"
/>
    </jb:bean>

    <jb:bean beanId="orderLine" class="example.entity.OrderLine"
createOnElement="order-item">
        <jb:value property="quantity" data="quantity" />
        <jb:wiring property="order" beanIdRef="order" />
        <jb:wiring property="product" beanIdRef="product" />
    </jb:bean>

    <dao:locator beanId="product" lookupOnElement="order-item"
onNoResult="EXCEPTION" uniqueResult="true">
        <dao:query>from Product p where p.id = :id</dao:query>
        <dao:params>
            <dao:value name="id" data="product" decoder="Integer"
/>
        </dao:params>
    </dao:locator>

    <dao:inserter beanId="order" insertOnElement="order" />

</smooks-resource-list>
```

5. If you want to use the named query **`productById`** instead of the query string, the DAO locator configuration will look like this:

```
<dao:locator beanId="product" lookupOnElement="order-item"
lookup="product.byId" onNoResult="EXCEPTION" uniqueResult="true">
    <dao:params>
```

```
        <dao:value name="id" data="product" decoder="Integer"/>
    </dao:params>
</dao:locator>
```

6.5. EXECUTING SMOOKS WITH A SESSIONREGISTER OBJECT

The following code executes Smooks. A **`SessionRegister`** object is used so the Hibernate Session can be accessed from within Smooks.

```
Smooks smooks = new Smooks("smooks-config.xml");

ExecutionContext executionContext = smooks.createExecutionContext();

// The SessionRegister provides the bridge between Hibernate and the
// Persistence Cartridge. We provide it with the Hibernate session.
// The Hibernate Session is set as default Session.
DaoRegister register = new SessionRegister(session);

// This sets the DAO Register in the executionContext for Smooks
// to access it.
PersistenceUtil.setDAORegister(executionContext, register);

Transaction transaction = session.beginTransaction();

smooks.filterSource(executionContext, source);

transaction.commit();
```

6.6. PERSISTING AN ORDER WITH DAO

1. The sample code below demonstrates how to persist an order with DAO. This example will read an XML file containing order information (this works the same for EDI, CSV, and so on). Using the Javabean cartridge, it will bind the XML data into a set of entity beans. It will locate the product entities and bind them to the order entity bean using the ID of the products within the order items (the product element). Finally, the order bean will be persisted.

   The order XML message looks like this:

   ```
   <order>
       <ordernumber>1</ordernumber>
       <customer>123456</customer>
       <order-items>
           <order-item>
               <product>11</product>
               <quantity>2</quantity>
           </order-item>
           <order-item>
               <product>22</product>
               <quantity>7</quantity>
           </order-item>
       </order-items>
   </order>
   ```

2. Use a custom DAO such as the example below to persist the Order entity:

```
@Dao
public class OrderDao {

    private final EntityManager em;

    public OrderDao(EntityManager em) {
        this.em = em;
    }

    @Insert
    public void insertOrder(Order order) {
        em.persist(order);
    }
}
```

When looking at this class you should notice the @Dao and @Insert annotations. The @Dao annotation declares that the OrderDao is a DAO object. The @Insert annotation declares that the insertOrder method should be used to insert Order entities.

3. Use a custom DAO as shown in the following example to lookup the Product entities:

```
@Dao
public class ProductDao {

    private final EntityManager em;

    public ProductDao(EntityManager em) {
        this.em = em;
    }

    @Lookup(name = "id")
    public Product findProductById(@Param("id")int id) {
        return em.find(Product.class, id);
    }
}
```

When looking at this class, notice the @Lookup and @Param annotation. The @Lookup annotation declares that the ProductDao.findProductById() method is used to lookup Product entities. The name parameter in the @Lookup annotation sets the lookup name reference for that method. When the name isn't declared, the method name will be used. The optional @Param annotation lets you name the parameters. This creates a better abstraction between Smooks and the DAO. If you don't declare the @Param annotation the parameters are resolved by their position.

4. When you have configured your order as shown above, the resulting Smooks configuration will look like this:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"

xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd"

xmlns:dao="http://www.milyn.org/xsd/smooks/persistence-1.2.xsd">

    <jb:bean BeanId="order" class="example.entity.Order"
```

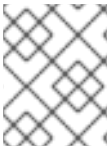
```
createOnElement="order">
        <jb:value property="ordernumber" data="ordernumber"/>
        <jb:value property="customerId" data="customer"/>
        <jb:wiring setterMethod="addOrderLine"
BeanIdRef="orderLine"/>
    </jb:bean>

    <jb:bean BeanId="orderLine" class="example.entity.OrderLine"
createOnElement="order-item">
        <jb:value property="quantity" data="quantity"/>
        <jb:wiring property="order" BeanIdRef="order"/>
        <jb:wiring property="product" BeanIdRef="product"/>
    </jb:bean>

    <dao:locator BeanId="product" dao="product" lookup="id"
lookupOnElement="order-item" onNoResult="EXCEPTION">
        <dao:params>
            <dao:value name="id" data="product" decoder="Integer"/>
        </dao:params>
    </dao:locator>

    <dao:inserter BeanId="order" dao="order"
insertOnElement="order"/>

</smooks-resource-list>
```

5. Use the following code to execute Smooks:

```
Smooks smooks=new Smooks("./smooks-configs/smooks-dao-config.xml");
ExecutionContext executionContext=smooks.createExecutionContext();

// The register is used to map the DAO's to a DAO name.
// The DAO name is used in the configuration.
// The MapRegister is a simple Map like implementation of the
DaoRegister.
DaoRegister<object>register = MapRegister.builder()
        .put("product",new ProductDao(em))
        .put("order",new OrderDao(em))
        .build();

PersistenceUtil.setDAORegister(executionContext,mapRegister);

// Transaction management from within Smooks isn't supported yet,
// so we need to do it outside the filter execution
EntityTransaction tx=em.getTransaction();
tx.begin();

smooks.filter(new StreamSource(messageIn),null,executionContext);

tx.commit();
```

CHAPTER 7. "GROOVY" SCRIPTING

7.1. GROOVY



NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

Groovy is an agile and dynamic language for the Java Virtual Machine that builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk.

Refer to http://groovy.codehaus.org/ for more information.

7.2. GROOVY EXAMPLE

Support for *Groovy* scripting is available through the *configuration namespace* (http://www.milyn.org/xsd/smooks/groovy-1.1.xsd.) This namespace provides support for DOM- and SAX-based scripting. See the example below:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:g="http://www.milyn.org/xsd/smooks/groovy-1.1.xsd">

    <g:groovy executeOnElement="xxx">
        <g:script>
        <!--
        //Rename the target fragment element from "xxx" to "yyy"...
        DomUtils.renameElement(element, "yyy", true, true);
        -->
        </g:script>
    </g:groovy>

</smooks-resource-list>
```

7.3. GROOVY TIPS

- The *visited element* is available to the script through the variable appropriately namedelement. (It is also available under the variable name which is equal to the element name but only if the name of the latter is limited alpha-numeric characters.)
- *Execute Before*/*Execute After*: by default, the script executes on the visitAfter event. Direct it to execute on the visitBefore by setting the executeBefore attribute to **`true`**.
- *Comment*/*CDATA Script Wrapping*: the script can be wrapped in an **`XML Comment`** or **`CDATA`** section if it contains special XML characters.

7.4. IMPORTS

Add *imports* using the imports element. A number of classes are automatically imported.

- **`org.milyn.xml.DomUtils`**

- **`org.milyn.javabean.context.BeanContext`**. Only in Smooks 1.3 and later.
- **`org.milyn.javabean.repository.BeanRepository`**
- **`org.w3c.dom.*`**
- **`groovy.xml.dom.DOMCategory`**
- **`groovy.xml.dom.DOMUtil`**
- **`groovy.xml.DOMBuilder`**

7.5. USING MIXED-DOM-AND-SAX WITH GROOVY

Groovy has support for the mixed-DOM-and-SAX model. You can use **Groovy**'s DOM utilities to process a targeted message fragment. A DOM "element" will be received by the **Groovy** script, even when the SAX filter is being used. This makes **Groovy** scripting using the SAX filter much easier while maintaining the ability to process huge messages in a streamed fashion.

7.6. MIXED-DOM-AND-SAX TIPS

Things to be careful of:

- it is only available in the default mode (that is, when executeBefore equals **`false`**). If executeBefore is configured to be **`true`**, this facility will not be available, which means that the **Groovy** script will only have access to SAXElements.
- writeFragment must be called in order to write the DOM fragment to a **`Smooks.filterSource StreamResult`**.
- a performance overhead will be incurred by using this DOM construction facility. (It can still process huge messages but it might take a slightly longer period of time. The compromise is between "usability" and performance.)

7.7. MIXED-DOM-AND-SAX EXAMPLE

Procedure 7.1. Task

1. Take an XML message such as the following sample:

```
<shopping>
    <category type="groceries">
        <item>Chocolate</item>
        <item>Coffee</item>
    </category>
    <category type="supplies">
        <item>Paper</item>
        <item quantity="4">Pens</item>
    </category>
        <category type="present">
        <item when="Aug 10">Kathryn's Birthday</item>
    </category>
</shopping>
```

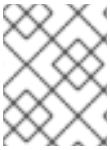
2. You can modify the "supplies" category in the above shopping list by adding two more "pens". To do this, write a simple **Groovy** script and aim it at the message's <category> elements.

3. As a result, the script simply iterates over the <item> elements in the category, and in instances where the category type is "supplies" and the item is "pens", the quantity is incremented by two:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"
   xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd"
   xmlns:g="http://www.milyn.org/xsd/smooks/groovy-1.1.xsd">

   <core:filterSettings type="SAX" />

   <g:groovy executeOnElement="category">
      <g:script>
      <!--
         use(DOMCategory) {
          // Modify "supplies": we need an extra 2 pens...
          if (category.'@type' == 'supplies') {
              category.item.each { item ->
                  if (item.text() == 'Pens') {
                      item['@quantity'] =
item.'@quantity'.toInteger() + 2;
                  }
              }
          }
      }

      // When using the SAX filter, we need to explicitly write the
fragment
      // to the result stream...
      writeFragment(category);
      -->
      </g:script>
   </g:groovy>

</smooks-resource-list>
```

CHAPTER 8. ROUTING OUTPUT DATA

8.1. OUTPUT DATA OPTIONS



NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

Smooks supports a number of different options when it comes to splitting and routing message fragments. The ability to split messages into fragments and route these fragments to different kinds of endpoints (files, JMS, etc.) is a very important capability. Smooks provides this along with the following features:

- **`Basic Fragment Splitting`**: A *dumb split* allows you to split fragments from a message and route them to a file, without performing any transformation on the split message fragments before routing. Basic splitting and routing involves defining the XPath of the message fragment to be split out and the defining a routing component (for example, JBoss ESB or Camel) to route that unmodified split message fragment.
- **`Complex Fragment Splitting`**: Basic fragment splitting works for many use cases and is what most splitting and routing solutions offer. Smooks extends the basic splitting capabilities by allowing you to perform transformations on the split fragment data before routing is applied. For example, merging in the customer-details order information with each order-item information before performing the routing order-item split fragment routing.
- **`In Stream Splitting and Routing`** (huge message support): Because Smooks can perform routing in stream (not batched up for routing after processing the complete message), it is able to accommodate processing of huge message streams that are gigabytes in size.
- **`Multiple Splitting and Routing`**: Conditionally split and route multiple message fragments (different formats XML, EDI, Java etc) to different endpoints in a single filtering pass of the input message stream. For example, routing an OrderItem Java object instance to the **`HighValueOrdersValidation`** JMS Queue for order items with a value greater than $1,000 and route all order items (unconditional) as XML/JSON to a HTTP endpoint for logging.

8.2. ROUTING WITH APACHE CAMEL

1. To route message fragments to Apache Camel endpoints, use the camel:route configuration from the http://www.milyn.org/xsd/smooks/camel-1.4.xsd configuration namespace.
2. Route to a Camel endpoint by specifying the following in your Smooks configuration:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"
         xmlns:camel="http://www.milyn.org/xsd/smooks/camel-
1.4.xsd">

  <!-- Create some bean instances from the input source... -->
  <jb:bean beanId="orderItem"  ... ">
    <!-- etc... See Smooks Java Binding docs -->
  </jb:bean>
```

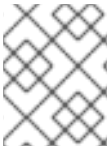
```
  <!-- Route bean to camel endpoints... -->
  <camel:route beanId="orderItem">
    <camel:to endpoint="direct:slow" if="orderItem.priority ==
'Normal'" />
    <camel:to endpoint="direct:express" if="orderItem.priority ==
'High'" />
  </camel:route>

</smooks-resource-list>
```

In the above example, Javabeans is routed from the Smooks BeanContext to the Camel Endpoints. You can also apply templates (such as FreeMarker) to these same beans and route the templating result instead of the beans (such as as XML and CSV).

`routeOnElement`.

CHAPTER 9. PERFORMANCE TUNING

9.1. PERFORMANCE TUNING TIPS



NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

Cache and reuse the Smooks object.

Initialization of Smooks takes some time and therefore it is important it is reused.

Pool reader instances where possible

This can result in a huge performance boost, as some readers are very expensive to create.

Use SAX filtering where possible

SAX processing is a lot faster than **DOM** processing and uses less memory. It is mandatory for processing large messages. Check that all of the Smooks cartridges are **SAX**-compatible.

Turn off debug logging

Smooks performs some intensive debug logging in parts of the code. This can result in significant additional processing overhead and lower throughput.

IMPORTANT

Remember that not having your logging configured at all may result in debug log statements being executed.

Only use the HTMLReportGenerator in a development environment.

When it has been enabled, the **HTMLReportGenerator** incurs a significant performance overhead and with large message, can even result in **OutOfMemory** exceptions.

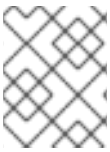
Contextual selectors

Contextual selectors can obviously have a negative effect on performance. For example, evaluating a match for a selector like **"a/b/c/d/e"** will obviously require more processing than that of a selector like **"d/e"**. Obviously there will be situations where your data model will require deep selectors, but where it does not, you should try to optimize your selectors for performance.

Where possible, avoid using the Virtual Bean Model and create beans instead of maps. Creating and adding data to Maps is a lot slower than creating "*plain old Java objects*" (**POJO**s) and calling the "setter" methods.

CHAPTER 10. TESTING

10.1. UNIT TESTING



NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

- To undertake unit testing with Smooks, follow the example below:

```
public class MyMessageTransformTest
{
    @Test
    public void test_transform() throws IOException, SAXException
    {
        Smooks smooks = new Smooks(
            getClass().getResourceAsStream("smooks-config.xml") );

        try {
            Source source = new StreamSource(
                getClass().getResourceAsStream("input-message.xml"
) );
            StringResult result = new StringResult();

            smooks.filterSource(source, result);

            // compare the expected xml with the transformation
result.
            XMLUnit.setIgnoreWhitespace( true );
            XMLAssert.assertXMLEqual(
                new InputStreamReader(
                getClass().getResourceAsStream("expected.xml")),
                new StringReader(result.getResult()));
        } finally {
            smooks.close();
        }
    }
}
```

The test case above uses a piece of software called **XMLUnit** (see http://xmlunit.sourceforge.net for more information.)

NOTE

The following **Maven** dependency was needed for the above test:

```
<dependency>
    <groupId>xmlunit</groupId>
    <artifactId>xmlunit</artifactId>
    <version>1.1</version>
</dependency>
```

CHAPTER 11. COMMON USE CASES

11.1. SUPPORT FOR PROCESSING HUGE MESSAGES

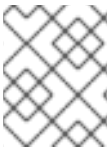
NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

Smooks supports the following types of processing for huge messages:

- **`One-to-one transformation`**: This is the process of transforming a huge message from its source format (for example, XML) to a huge message in a target format (EDI, CSV, XML, and so on).
- **`Splitting and routing`**: Splitting of a huge message into smaller (more consumable) messages in any format (EDI, XML, Java etc.) and routing of those smaller messages to a number of different destination types (File, JMS, Database).
- **`Persistence`**: Persisting the components of the huge message to a Database, from where they can be more easily queried and processed. Within Smooks, we consider this to be a form of splitting and routing (routing to a Database).

All of the above is possible without writing any code (that is, in a declarative manner). They can also be handled in a single pass over the source message, splitting and routing in parallel (plus routing to multiple destinations of different types and in different formats).



NOTE

When processing huge messages with Smooks, make sure you are using the SAX filter for better performance.

11.2. TRANSFORMING HUGE MESSAGES WITH FREEMARKER

To process a huge message by transforming it into a single message of another format, you can apply multiple FreeMarker templates to the Source message Event Stream and output it to a Smooks.filterSource Result stream. You can do this in one of two ways:

- Using FreeMarker and NodeModels for the model.
- Using FreeMarker and a Java Object model for the model. The model can be constructed from data in the message, using the Javabean Cartridge.

11.3. HUGE MESSAGES AND NODEMODELS

When a message is huge, you must identify its multiple NodeModels so that the runtime memory footprint is as low as possible. You cannot process the message using a single model because the full message is too big to hold in memory. In the case of the order message, there are two models: one for the main order data and one for the order-item data.

Most data that will be in memory at any one time is the main order data, plus one of the order-items. Because the NodeModels are nested, Smooks makes sure that the order data NodeModel never contains any of the data from the order-item NodeModels. Also, as Smooks filters the message, the

order-item NodeModel will be overwritten for every order-item (that is, they are not collected).

11.4. CONFIGURING SMOOKS TO CAPTURE MULTIPLE NODEMODELS

1. To configure Smooks to capture multiple NodeModels for use by the FreeMarker templates, you should configure the *DomModelCreator* visitor. It should be targeted at the root node of each model. Note again that Smooks also makes this available to SAX filtering (the key to processing huge messages).

   This is The Smooks configuration for creating the NodeModels for the message:

   ```
   <?xml version="1.0"?>
   <smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
   1.1.xsd"

   xmlns:core="http://www.milyn.org/xsd/smooks/smooks-core-1.3.xsd"

   xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

       <!--
       Filter the message using the SAX Filter (i.e. not DOM, so no
       intermediate DOM for the "complete" message - there are "mini"
   DOMs
       for the NodeModels below)....
       -->
       <core:filterSettings type="SAX" defaultSerialization="false" />

       <!--
       Create 2 NodeModels. One high level model for the "order"
       (header etc) and then one for the "order-item" elements...
       -->
       <resource-config selector="order,order-item">
           <resource>org.milyn.delivery.DomModelCreator</resource>
       </resource-config>

       <!-- FreeMarker templating configs to be added below... -->
   ```

2. Next, apply the following FreeMarker templates:

   - A template to output the order **header** details, up to but not including the order items.
   - A template for each of the order items, to generate the item elements in the salesorder.
   - A template to close out the message.

   With Smooks, you can implement this by defining two FreeMarker templates. One to cover points one and three (combined) above, and a second to cover the item elements.

3. Applt the first FreeMarker template. It is targeted at the order-items element and looks like this:

   ```
   <ftl:freemarker applyOnElement="order-items">
           <ftl:template><!--<salesorder>
       <details>
           <orderid>${order.@id}</orderid>
           <customer>
   ```

```
            <id>${order.header.customer.@number}</id>
            <name>${order.header.customer}</name>
        </customer>
    </details>
    <itemList>
    <?TEMPLATE-SPLIT-PI?>
    </itemList>
</salesorder>-->
        </ftl:template>
    </ftl:freemarker>
```

The **?TEMPLATE-SPLIT-PI?** processing instruction tells Smooks where to split the template, outputting the first part of the template at the start of the order-items element, and the other part at the end of the order-items element. The item element template (the second template) will be output in between.

4. Apply the second FreeMarker template. This outputs the item elements at the end of every order-item element in the source message:

```
<ftl:freemarker applyOnElement="order-item">
        <ftl:template><!-- <item>
    <id>${.vars["order-item"].@id}</id>
    <productId>${.vars["order-item"].product}</productId>
    <quantity>${.vars["order-item"].quantity}</quantity>
    <price>${.vars["order-item"].price}</price>
</item>-->
        </ftl:template>
    </ftl:freemarker>
</smooks-resource-list>
```

Because the second template fires on the end of the order-item elements, it effectively generates output into the location of the **?TEMPLATE-SPLIT-PI?** processing instruction in the first template. Note that the second template could have also referenced data in the **order** NodeModel.

5. Apply a closing template of your choice.

NOTE

This approach to performing a one-to-one transformation of a huge message works because the only objects in memory at any one time are the order header details and the current order-item details (in the Virtual Object Model). Obviously it can' work if the transformation is so obscure as to always require full access to all the data in the source message, for example if the messages needs to have all the order items reversed in order (or sorted). In such a case however, you do have the option of routing the order details and items to a database and then using the database's storage, query and paging features to perform the transformation.

11.5. MESSAGE SPLITTING REQUIREMENTS

You can process huge messages by splitting them into smaller messages that can be processed independently. Splitting and routing is sometimes also needed with smaller messages (message size may be irrelevant) where, for example, order items in an an order message need to be split out and

routed (based on content) to different departments or partners for processing. Under these conditions, the message formats required at the different destinations may also vary as shown in the examples below:

- **`destination1`**: required XML via the file system,
- **`destination2`**: requires Java objects via a JMS Queue,
- **`destination3`**: picks the messages up from a table in a Database etc.
- **`destination4`**: requires EDI messages via a JMS Queue,

You can perform multiple splitting and routing operations to multiple destinations (of different types) in a single pass over a message.

11.6. STREAMING SPLIT MESSAGES THROUGH SMOOKS

As you stream the message through Smooks:

- Repeatedly create a standalone message (split) for the fragment to be routed.
- Repeatedly bind the split message into the bean context under a unique beanId.
- Repeatedly route the split message to the required endpoint (whether it be a file, DB, JMS or ESB).

These operations happen for each instance of the split message found in the source message, for example, for each orderItem in an order message.

11.7. METHODS FOR CREATING SPLIT MESSAGES

- A basic (untransformed/unenriched) fragment split and bind. This serializes a message fragment (repeatedly) to its XML form and stores it in the bean context as a String.
- A more complex approach using the Java Binding and Templating Cartridges, where you configure Smooks to extract data from the source message and and into the bean context (using jb:bean configs) and then (optionally) apply templates to create the split messages. This has the following advantages:
  - Allows for transformation of the split fragments, that is, not just XML as with the basic option.
  - Allows for enrichment of the message.
  - Allows for more complex splits, with the ability to merge data from multiple source fragments into each split message, for example not just the orderItem fragments but the order header info too.
  - Allows for splitting and routing of Java Objects as the Split messages (for example, over JMS).

11.8. SERIALIZING MESSAGES

1. To split and route fragments of a message, use the basic frag:serialize and *:router components (jms:router, file:router and so on) from the Routing Cartridge. The frag:serialize component has its own configuration in the http://www.milyn.org/xsd/smooks/fragment-routing-1.2.xsd

namespace.

2. Use the example below for serializing the contents of a SOAP message body and storing it in the bean context under the beanId of soapBody:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:frag="http://www.milyn.org/xsd/smooks/fragment-
routing-1.2.xsd">

    <frag:serialize fragment="Envelope/Body" bindTo="soapBody"
childContentOnly="true"/>

</smooks-resource-list>
```

3. Use this code to execute it:

```
Smooks smooks = new Smooks(configStream);
JavaResult javaResult = new JavaResult();

smooks.filterSource(new StreamSource(soapMessageStream),
javaResult);

String bodyContent =
javaResult.getBean("soapBody").toString().trim();
```

4. To do this programatically, use this code:

```
Smooks smooks = new Smooks();

smooks.addVisitor(new FragmentSerializer().setBindTo("soapBody"),
"Envelope/Body");

JavaResult javaResult = new JavaResult();
smooks.filterSource(new StreamSource(soapMessageStream),
javaResult);

String bodyContent =
javaResult.getBean("soapBody").toString().trim();
```

11.9. ROUTING SPLIT MESSAGES EXAMPLE

The following is a quick example, showing the configuration for routing split messages (this time order-item fragments) to a JMS destination for processing:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:frag="http://www.milyn.org/xsd/smooks/fragment-routing-1.2.xsd"
xmlns:jms="http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd">

    <!-- Create the split messages for the order items... -->
    <frag:serialize fragment="order-items/order-item" bindTo="orderItem"
/>
```

```
    <!-- Route each order items split mesage to the orderItem JMS
processing queue... -->
    <jms:router routeOnElement="order-items/order-item" beanId="orderItem"
destination="orderItemProcessingQueue" />

</smooks-resource-list>
```

11.10. FILE-BASED ROUTING

File-based routing is performed via the file:outputStream configuration from the http://www.milyn.org/xsd/smooks/file-routing-1.1.xsd configuration namespace. You can combine the following Smooks functionality to split a message out into smaller messages on the file system.

11.11. FILE-BASED ROUTING COMPONENTS

Table 11.1. File-based Routing Components

Component	Description
The Javabean Cartridge	Extracts data from the message and holds it in variables in the bean context. You could also use DOM NodeModels for capturing the order and order-item data to be used as the templating data models.
file:outputStream	This configuration from the Routing Cartridge is used for managing file system streams (naming, opening, closing, throttling creation etc).
Templating Cartridge (FreeMarker Templates)	Used for generating the individual split messages from data bound in the bean context by the Javabean Cartridge (see first point above). The templating result is written to the file output stream (see second point above).

11.12. HUGE MESSAGE PROCESSING

Huge Message Processing

In the example, a huge order message needs to be sent while routing the individual order item details to file. The split messages contain data from the order header and root elements:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:core="http://www.milyn.org/xsd/smooks/smooks-
core-1.3.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-
1.4.xsd"
                      xmlns:file="http://www.milyn.org/xsd/smooks/file-
routing-1.1.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">
```

```
        <!--
        Filter the message using the SAX Filter (i.e. not DOM, so no
        intermediate DOM, so we can process huge messages...
        -->
        <core:filterSettings type="SAX" />

        <!-- Extract and decode data from the message. Used in the
freemarker template (below).
                Note that we could also use a NodeModel here... -->
(1)     <jb:bean beanId="order" class="java.util.Hashtable"
createOnElement="order">
            <jb:value property="orderId" decoder="Integer"
data="order/@id"/>
            <jb:value property="customerNumber" decoder="Long"
data="header/customer/@number"/>
            <jb:value property="customerName" data="header/customer"/>
            <jb:wiring property="orderItem" beanIdRef="orderItem"/>
        </jb:bean>
(2)     <jb:bean beanId="orderItem" class="java.util.Hashtable"
createOnElement="order-item">
            <jb:value property="itemId" decoder="Integer" data="order-
item/@id"/>
            <jb:value property="productId" decoder="Long" data="order-
item/product"/>
            <jb:value property="quantity" decoder="Integer" data="order-
item/quantity"/>
            <jb:value property="price" decoder="Double" data="order-
item/price"/>
        </jb:bean>

        <!-- Create/open a file output stream. This is writen to by the
freemarker template (below).. -->
(3)     <file:outputStream openOnElement="order-item"
resourceName="orderItemSplitStream">
            <file:fileNamePattern>order-${order.orderId}-
${order.orderItem.itemId}.xml</file:fileNamePattern>

<file:destinationDirectoryPattern>target/orders</file:destinationDirectory
Pattern>
            <file:listFileNamePattern>order-
${order.orderId}.lst</file:listFileNamePattern>

            <file:highWaterMark mark="10"/>
        </file:outputStream>

        <!--
        Every time we hit the end of an <order-item> element, apply this
freemarker template,
        outputting the result to the "orderItemSplitStream" OutputStream,
which is the file
        output stream configured above.
        -->
(4)     <ftl:freemarker applyOnElement="order-item">
            <ftl:template>target/classes/orderitem-
split.ftl</ftl:template>
            <ftl:use>
```

```
                <!-- Output the templating result to the
"orderItemSplitStream" file output stream... -->
                <ftl:outputTo
outputStreamResource="orderItemSplitStream"/>
            </ftl:use>
        </ftl:freemarker>

</smooks-resource-list>
```

Smooks Resource configurations shown in number one and two above define the Java Bindings for extracting the order header information (config #1) and the order-item information (config #2). When processing a huge message, make sure you only have the current order item in memory at any one time. The Smooks Javabean Cartridge manages all this for you, creating and recreating the orderItem beans as the order-item fragments are being processed.

The **`file:outputStream`** configuration in configuration number three manages the generation of the files on the file system. As you can see from the configuration, the file names can be dynamically constructed from data in the bean context. You can also see that it can throttle the creation of the files via the **`highWaterMark`** configuration parameter. This helps you manage file creation so as not to overwhelm the target file system.

Smooks Resource configuration number four defines the FreeMarker templating resource used to write the split messages to the OutputStream created by the **`file:outputStream`** (config #3). See how configuration 4 references the **`file:outputStream`** resource. The Freemarker template is as follows:

```
<orderitem id="${.vars["order-item"].@id}" order="${order.@id}">
    <customer>
        <name>${order.header.customer}</name>
        <number>${order.header.customer.@number}</number>
    </customer>
    <details>
        <productId>${.vars["order-item"].product}</productId>
        <quantity>${.vars["order-item"].quantity}</quantity>
        <price>${.vars["order-item"].price}</price>
    </details>
</orderitem>
```

11.13. JMS ROUTING

JMS Routing

JMS routing is performed via the jms:router configuration from the http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd configuration namespace. The following is an example jms:router configuration that routes an orderItem_xml bean to a JMS Queue named smooks.exampleQueue:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:core="http://www.milyn.org/xsd/smooks/smooks-
core-1.3.xsd"
                      xmlns:jms="http://www.milyn.org/xsd/smooks/jms-
routing-1.2.xsd"

xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

        <!--
```

```
        Filter the message using the SAX Filter (i.e. not DOM, so no
        intermediate DOM, so we can process huge messages...
        -->
        <core:filterSettings type="SAX" />

(1)     <resource-config selector="order,order-item">
            <resource>org.milyn.delivery.DomModelCreator</resource>
        </resource-config>

(2)     <jms:router routeOnElement="order-item" beanId="orderItem_xml"
destination="smooks.exampleQueue">
            <jms:message>
                <!-- Need to use special FreeMarker variable ".vars" -->
                <jms:correlationIdPattern>${order.@id}-${.vars["order-
item"].@id}</jms:correlationIdPattern>
            </jms:message>
            <jms:highWaterMark mark="3"/>
        </jms:router>

(3)     <ftl:freemarker applyOnElement="order-item">
            <!--
            Note in the template that we need to use the special
FreeMarker variable ".vars"
            because of the hyphenated variable names ("order-item"). See
http://freemarker.org/docs/ref_specvar.html.
            -->
            <ftl:template>/orderitem-split.ftl</ftl:template>
            <ftl:use>
                <!-- Bind the templating result into the bean context,
from where
                it can be accessed by the JMSRouter (configured above). -
->
                <ftl:bindTo id="orderItem_xml"/>
            </ftl:use>
        </ftl:freemarker>

</smooks-resource-list>
```

In this case, we route the result of a FreeMarker templating operation to the JMS Queue (that is, as a String). We could also have routed a full Object Model, in which case it would be routed as a Serialized ObjectMessage.

11.14. ROUTING TO A DATABASE

1. To route an order and order item data to a database, you should define a set of Java bindings that extract the order and order-item data from the data stream:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"

xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <!-- Extract the order data... -->
    <jb:bean beanId="order" class="java.util.Hashtable"
```

```
createOnElement="order">
        <jb:value property="orderId" decoder="Integer"
data="order/@id"/>
        <jb:value property="customerNumber" decoder="Long"
data="header/customer/@number"/>
        <jb:value property="customerName" data="header/customer"/>
    </jb:bean>

    <!-- Extract the order-item data... -->
    <jb:bean beanId="orderItem" class="java.util.Hashtable"
createOnElement="order-item">
        <jb:value property="itemId" decoder="Integer" data="order-
item/@id"/>
        <jb:value property="productId" decoder="Long" data="order-
item/product"/>
        <jb:value property="quantity" decoder="Integer" data="order-
item/quantity"/>
        <jb:value property="price" decoder="Double" data="order-
item/price"/>
    </jb:bean>
```

2. Next you need to define datasource configuration and a number of db:executor configurations that will use that datasource to insert the data that was bound into the Java Object model into the database. This is the datasource configuration (namespace http://www.milyn.org/xsd/smooks/datasource-1.3.xsd) for retrieving a direct database connection:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:ds="http://www.milyn.org/xsd/smooks/datasource-
1.3.xsd">

    <ds:direct bindOnElement="#document"
            datasource="DBExtractTransformLoadDS"
            driver="org.hsqldb.jdbcDriver"
            url="jdbc:hsqldb:hsql://localhost:9201/milyn-hsql-9201"
            username="sa"
            password=""
            autoCommit="false" />

</smooks-resource-list>
```

3. It is possible to use a JNDI datasource for retrieving a database connection:

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd" xmlns:ds="http://www.milyn.org/xsd/smooks/datasource-
1.3.xsd">

    <!-- This JNDI datasource can handle JDBC and JTA transactions
or
          it can leave the transaction managment to an other
external component.
          An external component could be an other Smooks visitor,
the EJB transaction manager
```

```
            or you can do it your self. -->
    <ds:JNDI
        bindOnElement="#document"
        datasource="DBExtractTransformLoadDS"
        datasourceJndi="java:/someDS"
        transactionManager="JTA"
        transactionJndi="java:/mockTransaction"
        targetProfile="jta"/>

</smooks-resource-list>
```

4. The datasource schema describes and documents how you can configure the datasource. This is the db:executor configuration (namespace http://www.milyn.org/xsd/smooks/db-routing-1.1.xsd):

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd"
                      xmlns:db="http://www.milyn.org/xsd/smooks/db-
routing-1.1.xsd">

    <!-- Assert whether it's an insert or update. Need to do this
just before we do the insert/update... -->
    <db:executor executeOnElement="order-items"
datasource="DBExtractTransformLoadDS" executeBefore="true">
        <db:statement>select OrderId from ORDERS where OrderId =
${order.orderId}</db:statement>
        <db:resultSet name="orderExistsRS"/>
    </db:executor>

    <!-- If it's an insert (orderExistsRS.isEmpty()), insert the
order before we process the order items... -->
    <db:executor executeOnElement="order-items"
datasource="DBExtractTransformLoadDS" executeBefore="true">
        <condition>orderExistsRS.isEmpty()</condition>
        <db:statement>INSERT INTO ORDERS VALUES(${order.orderId},
${order.customerNumber}, ${order.customerName})</db:statement>
    </db:executor>

    <!-- And insert each orderItem... -->
    <db:executor executeOnElement="order-item"
datasource="DBExtractTransformLoadDS" executeBefore="false">
        <condition>orderExistsRS.isEmpty()</condition>
        <db:statement>INSERT INTO ORDERITEMS VALUES
(${orderItem.itemId}, ${order.orderId}, ${orderItem.productId},
${orderItem.quantity}, ${orderItem.price})</db:statement>
    </db:executor>

    <!-- Ignoring updates for now!! -->

</smooks-resource-list>
```

CHAPTER 12. EXTENDING SMOOKS

12.1. APIS IN SMOOKS

NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

APIs

All existing Smooks functionality (Java Binding, EDI processing etc) is built through the extension of a number of well defined APIs.

The main extension points/APIs in Smooks are Reader and Visitor APIs:

Reader APIs

Those for processing Source/Input data (Readers) so as to make it consumable by other Smooks components as a series of well defined hierarchical events (based on the SAX event model) for all of the message fragments and sub-fragments.

Visitor APIs

Those for consuming the message fragment SAX Events produced by a Source/Input Reader.

12.2. CONFIGURING SMOOKS COMPONENTS

All Smooks components are configured in exactly the same way. When using the Smooks Core code, all Smooks components are **`resources`** which are configured using a **`SmooksResourceConfiguration`** instance.

12.3. NAMESPACE-SPECIFIC CONFIGURATIONS

Smooks provides mechanisms for constructing namespace (XSD) specific XML configurations for components. The most basic configuration (and the one that maps directly to the **`SmooksResourceConfiguration`** class) is the basic <resource-config> XML configuration from the base configuration namespace (http://www.milyn.org/xsd/smooks-1.1.xsd).

12.4. NAMESPACE-SPECIFIC CONFIGURATION EXAMPLE

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

    <resource-config selector="">
        <resource></resource>
        <param name=""></param>
    </resource-config>

</smooks-resource-list>
```

- The **`selector`** attribute is the mechanism by which the resource is "selected" (for example, it can be an XPath for a Visitor implementation).

- The **resource** element is the actual resource. This can be a Java Class name or some other form of resource such as a template. The resource is assumed to be a Java class name for the remainder for this section.
- The **param** elements are configuration parameters for the resource defined in the resource element.

12.5. RUNTIME REPRESENTATION

Smooks takes care of all the details of creating the runtime representation of the resource (for example, constructing the class named in the the resource element) and injects all the configuration parameters. It also works out what the resource type is, and from that, how to interpret things like the selector. (For example, if the resource is a Visitor instance, it knows the selector is an XPath, selecting a Source message fragment.)

12.6. CONFIGURATION ANNOTATIONS

After your component has been created, you need to configure it with the **<param>** element details. This is done using the **@ConfigParam** and **@Config** annotations.

12.7. THE @CONFIGPARAM ANNOTATION

The **@ConfigParam** annotation reflectively injects the named parameter from the **<param>** elements that have the same name as the annotated property itself. The name can be different but the default behavior matches against the name of the component property.

12.8. @CONFICPARAM BENEFITS

This annotation eliminates excess code from your component because it:

- Handles decoding of the <param> value before setting it on the annotated component property. Smooks provides DataDecoders for all of the main types (int, Double, File, Enums etc), but you can implement and use a custom DataDecoder where the out of the box decoders don't cover specific decoding requirements (for example, **@ConfigParam(decoder = MyQuirkyDataDecoder.class)**). Smooks will automatically use your custom decoder (that is, you won't need to define the decoder property on this annotation) if it is registered. See the DataDecoder Javadocs for details on registering a DataDecoder implementation such that Smooks will automatically locate it for decoding a specific data type.
- Supports a **choice** constraint for the **config** property, generating a configuration exception where the configured value is not one of the defined choice values. For example, you may have a property which has a constrained value set of **ON** and **OFF**. You can use the choice property on this annotation to constrain the config, raise exceptions, and so on. (For example, **@ConfigParam(choice = {"ON", "OFF"})**.)
- Can specify default config values e.g. **@ConfigParam(defaultVal = "true")**.
- Can specify whether or not the property config value is required or optional e.g. **@ConfigParam(use = Use.OPTIONAL)**. By default, all properties are **REQUIRED**, but setting a **defaultVal** implicitly marks the property as being **OPTIONAL**.

12.9. USING THE @CONFIGPARAM ANNOTATION

This example show the annotated component **DataSeeder** and its corresponding Smooks configuration:

```
public class DataSeeder
{
   @ConfigParam
   private File seedDataFile;

   public File getSeedDataFile()
   {
      return seedDataFile;
   }

   // etc...
}
```

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">
   <resource-config selector="dataSeeder">
      <resource>com.acme.DataSeeder</resource>
         <param name="seedDataFile">./seedData.xml</param>
      </resource-config>
</smooks-resource-list>
```

12.10. THE @CONFIG ANNOTATION

The **@Config** annotation reflectively injects the full **SmooksResourceConfiguration** instance, associated with the component resource, onto the annotated component property. An error will result if this annotation is added to a component property that is not of type **SmooksResourceConfiguration**.

12.11. USING THE @CONFIG ANNOTATION

```
public class MySmooksComponent
{
    @Config
    private SmooksResourceConfiguration config;

    // etc...
```

12.12. @INITIALIZE AND @UNINITIALIZE

Sometimes your component needs more involved configuration for which we need to write some "initialization" code. For this, Smooks provides the **@Initialize** annotation.

Likewise, there are times when you need to undo work performed during initialization when the associated Smooks instance is being discarded (garbage collected) e.g. to release some resources acquired during initialization. For this, Smooks provides the **@Uninitialize** annotation.

12.13. A BASIC INITIALIZATION/UN-INITIALIZATION SEQUENCE

This is a basic initialization/un-initialization sequence:

```
smooks = new Smooks(..);
```

```
// Initialize all annotated components
@Initialize

// Use the smooks instance through a series of filterSource invocations...
smooks.filterSource(...);
smooks.filterSource(...);
smooks.filterSource(...);
... etc ...

smooks.close();

// Uninitialize all annotated components
@Uninitialize
```

12.14. USING @INITIALIZE AND @UNINITIALIZE

Overview

In this example, assume we have a component that opens multiple connections to a database on initialization and then needs to release all those database resources when we close the Smooks instance.

```
public class MultiDataSourceAccessor
{
    @ConfigParam
    private File dataSourceConfig;

    Map<String, Datasource> datasources = new HashMap<String, Datasource>
();

    @Initialize
    public void createDataSources()
    {
        // Add DS creation code here....
        // Read the dataSourceConfig property to read the DS configs...
    }

    @Uninitialize
    public void releaseDataSources()
    {
        // Add DS release code here....
    }

    // etc...
}
```

When using the **@Initialize** and **@Uninitialize** annotations above, the following should be noted:

- The **@Initialize** and **@Uninitialize** methods must be public, zero-arg methods.
- The **@ConfigParam** properties are all initialized before the first **@Initialize** method is called. Therefore, you can use the **@ConfigParam** component properties as input to the initialization process.

- The **`@Uninitialize`** methods are all called in response to a call to the **`Smooks.close`** method.

12.15. DEFINING CUSTOM CONFIGURATION NAMESPACES

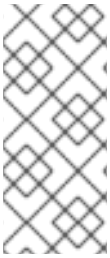
Smooks supports a mechanism for defining custom configuration namespaces for components. This allows you to support custom, XSD-based, configurations for your components that can be validated instead of treating them all as generic Smooks resources using the **`<resource-config>`** base configuration.

12.16. USING CUSTOM CONFIGURATION NAMESPACES

The basic process involves two steps:

1. Writing an configuration XSD for your component that extends the base http://www.milyn.org/xsd/smooks-1.1.xsd configuration namespace. This XSD must be supplied on the classpath with your component. It must be located in the **`/META-INF/`** folder and have the same path as the namespace URI. For example, if your extended namespace URI is http://www.acme.com/schemas/smooks/acme-core-1.0.xsd, then the physical XSD file must be supplied on the class-path in **`/META-INF/schemas/smooks/acme-core-1.0.xsd`**.

2. Writing a Smooks configuration namespace mapping configuration file that maps the custom name-space configuration into a **`SmooksResourceConfiguration`** instance. This file must be named (by convention) based on the name of the name-space it is mapping and must be physically located on the class-path in the same folder as the XSD. Extending the above example, the Smooks mapping file would be **`/META-INF/schemas/smooks/acme-core-1.0.xsd-smooks.xml`**. Note the **`-smooks.xml`** postfix.



NOTE

The easiest way to get familiar with this mechanism is by looking at existing extended namespace configurations within the Smooks code itself. All Smooks components (including the Java Binding functionality) use this mechanism for defining their configurations. Smooks Core itself defines a number of extended configuration namespaces.

12.17. IMPLEMENTING A SOURCE READER

You can implement a source reader for your custom data format which immediately opens all Smooks capabilities to that data format such as Java Binding, Templating, Persistence, Validation, Splitting, Routing etc. The only Smooks requirement is that the Reader implements the standard **`org.xml.sax.XMLReader`** interface from the Java JDK. However, if you want to be able to configure the Reader implementation, it needs to implement the **`org.milyn.xml.SmooksXMLReader`** interface. **`org.milyn.xml.SmooksXMLReader`** is an extension of **`org.xml.sax.XMLReader`**. You can easily use an existing **`org.xml.sax.XMLReader`** implementation, or implement a new one.

Refer to http://java.sun.com/j2se/1.5.0/docs/api/org/xml/sax/XMLReader.html for more details.

12.18. IMPLEMENTING A SOURCE READER FOR USE WITH SMOOKS

1. You should first implement a basic reader class as shown below:

```
public class MyCSVReader implements SmooksXMLReader
```

```
{
   // Implement all of the XMLReader methods...
}
```

Two methods from the **`org.xml.sax.XMLReader`** interface are of particular interest:

1. **`setContentHandler(ContentHandler)`** is called by Smooks Core. It sets the **`org.xml.sax.ContentHandler`** instance for the reader. The **`org.xml.sax.ContentHandler`** instance methods are called from inside the **`parse(InputSource)`** method.

2. **`parse(InputSource)`** : This is the method that receives the Source data input stream, parses it (i.e. in the case of this example, the CSV stream) and generates the SAX event stream through calls to the **`org.xml.sax.ContentHandler`** instance supplied in the **`setContentHandler(ContentHandler)`** method.

Refer to http://download.oracle.com/javase/6/docs/api/org/xml/sax/ContentHandler.html for more details.

2. Configure your CSV reader with the names of the fields associated with the CSV records. Configuring a custom reader implementation is the same for any Smooks component. See the example below:

```
public class MyCSVReader implements SmooksXMLReader
{

    private ContentHandler contentHandler;

    @ConfigParam
    private String[] fields; // Auto decoded and injected from the
"fields" <param> on the reader config.

    public void setContentHandler(ContentHandler contentHandler) {
        this.contentHandler = contentHandler;
    }

    public void parse(InputSource csvInputSource) throws
IOException, SAXException {
        // TODO: Implement parsing of CSV Stream...
    }

    // Other XMLReader methods...
}
```

3. Now that you have the basic Reader implementation stub, you can start writing unit tests to test the new reader implementation. To do this you will need something with CSV input. Observe the example below featuring a simple list of names in a file with the name **`names.csv`**:

```
Tom,Jones
Mike,Jones
Mark,Jones
```

4. Use a test Smooks configuration to configure Smooks with your MyCSVReader. As stated before, everything in Smooks is a resource and can be configured with the basic **`<resource-config>`** configuration. While this works fine, it's a little noisy, so Smooks provides a basic

<reader> configuration element specifically for the purpose of configuring a reader. The configuration for the test looks like the following, in the **mycsvread-config.xml**:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd">
   <reader class="com.acme.MyCSVReader">
      <params>
         <param name="fields">firstname,lastname</param>
      </params>
   </reader>
</smooks-resource-list>
```

5. Implement the JUnit test class:

```
public class MyCSVReaderTest extends TestCase
{

    public void test() {
        Smooks smooks = new
Smooks(getClass().getResourceAsStream("mycsvread-config.xml"));
        StringResult serializedCSVEvents = new StringResult();

        smooks.filterSource(new
StreamSource(getClass().getResourceAsStream("names.csv")),
serializedCSVEvents);

        System.out.println(serializedCSVEvents);

        // TODO: add assertions etc
    }
}
```

6. Implement the **parse** method:

```
public class MyCSVReader implements SmooksXMLReader
{
    private ContentHandler contentHandler;

    @ConfigParam
    private String[] fields; // Auto decoded and injected from the
"fields" <param> on the reader config.

    public void setContentHandler(ContentHandler contentHandler)
    {
        this.contentHandler = contentHandler;
    }

    public void parse(InputSource csvInputSource) throws
IOException, SAXException
    {
        BufferedReader csvRecordReader = new
BufferedReader(csvInputSource.getCharacterStream());
        String csvRecord;

        // Send the start of message events to the handler...
```

```
        contentHandler.startDocument();
        contentHandler.startElement(XMLConstants.NULL_NS_URI,
"message-root", "", new AttributesImpl());

        csvRecord = csvRecordReader.readLine();
        while(csvRecord != null)
        {
            String[] fieldValues = csvRecord.split(",");

            // perform checks...

            // Send the events for this record...
            contentHandler.startElement(XMLConstants.NULL_NS_URI,
"record", "", new AttributesImpl());
            for(int i = 0; i < fields.length; i++)
            {

contentHandler.startElement(XMLConstants.NULL_NS_URI, fields[i], "",
new AttributesImpl());

contentHandler.characters(fieldValues[i].toCharArray(), 0,
fieldValues[i].length());
                contentHandler.endElement(XMLConstants.NULL_NS_URI,
fields[i], "");
            }
            contentHandler.endElement(XMLConstants.NULL_NS_URI,
"record", "");

            csvRecord = csvRecordReader.readLine();
        }

        // Send the end of message events to the handler...
        contentHandler.endElement(XMLConstants.NULL_NS_URI,
"message-root", "");
        contentHandler.endDocument();
    }

    // Other XMLReader methods...
}
```

7. Run the unit test class to see the following output on the console (formatted):

```
<message-root>
    <record>
        <firstname>Tom</firstname>
        <lastname>Jones</lastname>
    </record>
    <record>
        <firstname>Mike</firstname>
        <lastname>Jones</lastname>
    </record>
    <record>
        <firstname>Mark</firstname>
        <lastname>Jones</lastname>
    </record>
</message-root>
```

-

After this, it is a case of expanding the tests, hardening the reader implementation code, and so on. Then you can use your reader to perform all sorts of operations supported by Smooks.

12.19. CONFIGURING THE READER WITH JAVA-BINDING-CONFIG.XML EXAMPLE

The following configuration (**`java-binding-config.xml`**) can be used to bind the names into a **`List`** of **`PersonName`** objects:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">
    <reader class="com.acme.MyCSVReader">
        <params>
         <param name="fields">firstname,lastname</param>
        </params>
    </reader>
    <jb:bean beanId="peopleNames" class="java.util.ArrayList"
createOnElement="message-root">
        <jb:wiring beanIdRef="personName" />
    </jb:bean>
    <jb:bean beanId="personName" class="com.acme.PersonName"
createOnElement="message-root/record">
        <jb:value property="first" data="record/firstname" />
        <jb:value property="last" data="record/lastname" />
    </jb:bean>
</smooks-resource-list>
```

Here is a test for that configuration:

```
public class MyCSVReaderTest extends TestCase
{
    public void test_java_binding()
    {
        Smooks smooks = new Smooks(getClass().getResourceAsStream("java-
binding-config.xml"));
        JavaResult javaResult = new JavaResult();

        smooks.filterSource(new
StreamSource(getClass().getResourceAsStream("names.csv")), javaResult);

        List<PersonName> peopleNames = (List<PersonName>)
javaResult.getBean("peopleNames");

        // TODO: add assertions etc
    }
}
```

12.20. TIPS FOR USING A READER

- Reader instances are never used concurrently. Smooks Core will create a new instance for every message, or, will pool and reuse instances as per the **`readerPoolSizeFilterSettings`** property.

- If your Reader requires access to the Smooks **`ExecutionContext`** for the current filtering context, your Reader needs to implement the **`org.milyn.xml.SmooksXMLReader`** interface.
- If your Source data is a binary data stream your Reader must implement the **`org.milyn.delivery.StreamReader`** interface.
- You can configure your reader within your source code (e.g. in your unit tests) using a **`GenericReaderConfigurator`** instance, which you then set on the **`Smooks`** instance.
- While the basic <reader> configuration is fine, it is possible to define a custom configuration namespace (XSD) for your custom CSV Reader implementation. This topic is not covered here. Review the source code to see the extended configuration namespace for the Reader implementations supplied with Smooks, e.g. the **`EDIReader`**, **`CSVReader`**, **`JSONReader`** etc. From this, you should be able to work out how to do this for your own custom Reader.

12.21. BINARY SOURCE READERS

A *binary source reader* is a reader for a binary data source. Your reader should implement the **`org.milyn.delivery.StreamReader`** interface. This is just a marker interface that tells the Smooks runtime to ensure that an **`InputStream`** is supplied.

The binary Reader implementation is essentially the same as a non-binary Reader implementation (see above), except that the implementation of the **`parse`** method should use the **`InputStream`** from the **`InputSource`** (i.e. call **`InputSource.getByteStream()`** instead of **`InputSource.getCharacterStream()`**) and generate the XML events from the decoded binary data.

12.22. IMPLEMENTING A BINARY SOURCE READER

1. To implement a binary source reader, observe the following **`parse`** method implementation:

```
public static class BinaryFormatXXReader implements SmooksXMLReader,
StreamReader
{
    @ConfigParam
    private String xProtocolVersion;

    @ConfigParam
    private int someOtherXProtocolConfig;

    // etc...

    public void parse(InputSource inputSource) throws IOException,
SAXException {
        // Use the InputStream (binary) on the InputSource...
        InputStream binStream = inputSource.getByteStream();

        // Create and configure the data decoder...
        BinaryFormatXDecoder xDecoder = new BinaryFormatXDecoder();
        xDecoder.setProtocolVersion(xProtocolVersion);

xDecoder.setSomeOtherXProtocolConfig(someOtherXProtocolConfig);
        xDecoder.setXSource(binStream);

        // Generate the XML Events on the contentHandler...
```

```
        contentHandler.startDocument();

        // Use xDecoder to fire startElement, endElement etc events
on the contentHandler (see previous section)...

        contentHandler.endDocument();
    }

    // etc....
}
```

2. Configure the **`BinaryFormatXXReader`** reader in your Smooks configuration as you would any other reader:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-
1.1.xsd">

    <reader class="com.acme.BinaryFormatXXReader">
        <params>
         <param name="xProtocolVersion">2.5.7</param>
         <param name="someOtherXProtocolConfig">1</param>
            ... etc...
        </params>
    </reader>

    ... Other Smooks configurations e.g. <jb:bean> configs for
binding the binary data into Java objects...

</smooks-resource-list>
```

3. Run the Smooks execution code (note the **`InputStream`** supplied to the **`StreamSource`**). In this case, two results are generated: XML and Java objects.

```
StreamResult xmlResult = new StreamResult(xmlOutWriter);
JavaResult javaResult = new JavaResult();

InputStream xBinaryInputStream = getXByteStream();

smooks.filterSource(new StreamSource(xBinaryInputStream), xmlResult,
javaResult);

// etc... Use the beans in the javaResult...
```

12.23. VISITOR IMPLEMENTATIONS

Visitor implementations are the workhorses of Smooks. Most of the out-of-the-box functionality in Smooks (Java Binding, Templating, Persistence, and so on) was created by using one or more Visitor implementations. Visitor implementations often collaborate through the **`ExecutionContext`** and **`ApplicationContext`** context objects, accomplishing a common goal by working together.

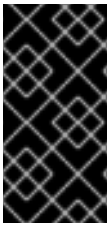
12.24. SUPPORTED VISITOR IMPLEMENTATIONS

1. SAX-based implementations based on the **`org.milyn.delivery.sax.SAXVisitor`** sub-interfaces.

2. DOM-based implementations based on the **`org.milyn.delivery.dom.DOMVisitor`** sub-interfaces.

12.25. SAX AND DOM VISITOR IMPLEMENTATIONS

Your implementation can support both SAX and DOM, but Red Hat recommends implementing a SAX only Visitor. SAX-based implementations are usually easier to create and perform faster.



IMPORTANT

All Visitor implementations are treated as stateless objects. A single Visitor instance must be usable concurrently across multiple messages, that is, across multiple concurrent calls to the **`Smooks.filterSource`** method. All state associated with the current **`Smooks.filterSource`** execution must be stored in the **`ExecutionContext`**.

12.26. THE SAX VISITOR API

The SAX Visitor API is made up of a number of interfaces. These interfaces are based on the **`org.xml.sax.ContentHandler`** SAX events that a **`SAXVisitor`** implementation can capture and processes. Depending on the use case being solved with the **`SAXVisitor`** implementation, you may need to implement one or all of these interfaces.

12.27. SAX VISITOR API INTERFACES

`org.milyn.delivery.sax.SAXVisitBefore`

Captures the **`startElement`** SAX event for the targeted fragment element:

```
public interface SAXVisitBefore extends SAXVisitor
{
    void visitBefore(SAXElement element, ExecutionContext
executionContext)
                                          throws SmooksException,
IOException;
}
```

`org.milyn.delivery.sax.SAXVisitChildren`

Captures the character based SAX events for the targeted fragment element, as well as Smooks generated (pseudo) events corresponding to the **`startElement`** events of child fragment elements:

```
public interface SAXVisitChildren extends SAXVisitor
{
    void onChildText(SAXElement element, SAXText childText,
ExecutionContext
                        executionContext) throws SmooksException,
IOException;

    void onChildElement(SAXElement element, SAXElement childElement,
```

```
          ExecutionContext executionContext) throws SmooksException,
IOException;
}
```

`org.milyn.delivery.sax.SAXVisitAfter`

Captures the **`endElement`** SAX event for the targeted fragment element:

```
 public interface SAXVisitAfter extends SAXVisitor
{
    void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                          throws SmooksException,
IOException;
}
```

12.28. SAX VISITOR API EXAMPLE

Illustrating API events using XML

This pulls together three interfaces into a single interface in the **`org.milyn.delivery.sax.SAXElementVisitor`** interface:

```
<message>
    <target-fragment>      <!-- SAXVisitBefore.visitBefore -->
        Text!!             <!-- SAXVisitChildren.onChildText -->
        <child>            <!-- SAXVisitChildren.onChildElement -->
        </child>
    </target-fragment>     <!-- SAXVisitAfter.visitAfter -->
</message>
```

The above is an illustration of a Source message event stream as XML. It could be EDI, CSV, JSON, or some other format. Consider it to be a Source message event stream, serialized as XML for easy reading.

As can be seen from the above SAX interfaces, the **`org.milyn.delivery.sax.SAXElement`** type is passed in all method calls. This object contains details about the targeted fragment element, including attributes and their values. It also contains methods for managing text accumulation, as well as accessing the **`Writer`** associated with any **`StreamResult`** instance that may have been passed in the **`Smooks.filterSource(Source, Result)`** method call.

12.29. TEXT ACCUMULATION WITH SAX

SAX is a stream based processing model. It doesn't create a Document Object Model (DOM) or "accumulate" event data in any way. This is why it is a suitable processing model for processing huge message streams.

12.30. ORG.MILYN.DELIVERY.SAX.SAXELEMENT

The **`org.milyn.delivery.sax.SAXElement`** will always contain attribute data associated with the targeted element, but will not contain the fragment child text data, whose SAX events (**`SAXVisitChildren.onChildText`**) occur between the **`SAXVisitBefore.visitBefore`** and **`SAXVisitAfter.visitAfter`** events. The text events are not accumulated on the **`SAXElement`**

because that could result in a significant performance drain. The downside to this is that if the **SAXVisitor** implementation needs access to the text content of a fragment, you need to explicitly tell Smooks to accumulate text for the targeted fragment. This is done by calling the **SAXElement.accumulateText** method from inside the **SAXVisitBefore.visitBefore** method implementation of your **SAXVisitor**.

12.31. TEXT ACCUMULATION EXAMPLE

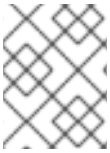
```
public class MyVisitor implements SAXVisitBefore, SAXVisitAfter
{
   public void visitBefore(SAXElement element, ExecutionContext
executionContext)
                                                throws SmooksException,
IOException
   {
      element.accumulateText();
   }

   public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                throws SmooksException,
IOException
   {
      String fragmentText = element.getTextContent();

      // ... etc ...
   }
}
```

12.32. THE @TEXTCONSUMER ANNOTATION

The **@TextConsumer** annotation can be used to annotate your **SAXVisitor** implementation instead of using the **SAXVisitBefore.visitBefore** method.

12.33. @TEXTCONSUMER EXAMPLE

```
@TextConsumer
public class MyVisitor implements SAXVisitAfter
{
   public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                throws SmooksException,
IOException
   {
      String fragmentText = element.getTextContent();

        // ... etc ...
    }
}
```



NOTE

The entirity of the fragment text will not be available until after the **`SAXVisitAfter.visitAfter`** event.

12.34. STREAMRESULT WRITING/SERIALIZATION

Overview

The **`Smooks.filterSource(Source, Result)`** method can take one or more of a number of different **`Result`** type implementations, one of which is the **`javax.xml.transform.stream.StreamResult`** class. Smooks streams the Source in and back out again through the **`StreamResult`** instance.

By default, Smooks will always serialize the full Source event stream as XML to any **`StreamResult`** instance provided to the **`Smooks.filterSource(Source, Result)`** method. If the Source provided to the **`Smooks.filterSource(Source, Result)`** method is an XML stream and a **`StreamResult`** instance is provided as one of the **`Result`** instances, the Source XML will be written out to the **`StreamResult`** unmodified, unless the Smooks instance is configured with one or more **`SAXVisitor`** implementations that modify one or more fragments.

12.35. CONFIGURING STREAMRESULT WRITING/SERIALIZATION

1. To turn the default serialization behavior on or off, access the filter settings and configure them to do so.

2. To modify the serialized form of one of the message fragments, implement a **`SAXVisitor`** to perform the transformation and target it at the message fragment using an XPath-like expression.

3. To modify the serialized form of a message fragment, use one of the provided templating components. These components are also **`SAXVisitor`** implementations.

12.36. IMPLEMENTING THE SAXVISITOR

1. To implement a **`SAXVisitor`** geared towards transforming the serialized form of a fragment, program Smooks so the **`SAXVisitor`** implementation will be writing to the **`StreamResult`**. This is because Smooks supports targeting of multiple **`SAXVisitor`** implementations at a single fragment, but only one **`SAXVisitor`** is allowed to write to the **`StreamResult`**, per fragment.

2. If a second **`SAXVisitor`** attempts to write to the **`StreamResult`**, a **`SAXWriterAccessException`** will result and you will need to modify your Smooks configuration.

3. To specify the **`StreamResult`** to write, the **`SAXVisitor`** needs to "acquire ownership" of the **`Writer`** to the **`StreamResult`**. It does this by making a call to the **`SAXElement.getWriter(SAXVisitor)`** method from inside the **`SAXVisitBefore.visitBefore`** methods implementation, passing **`this`** as the **`SAXVisitor`** parameter.

12.37. SAXVISITOR IMPLEMENTATION EXAMPLE

```
public class MyVisitor implements SAXElementVisitor
```

```
{
   public void visitBefore(SAXElement element, ExecutionContext
executionContext)
                                                throws SmooksException,
IOException
   {
      Writer writer = element.getWriter(this);

      // ... write the start of the fragment...
   }

   public void onChildText(SAXElement element, SAXText childText,
                                    ExecutionContext executionContext)
                                       throws SmooksException,
IOException
   {
      Writer writer = element.getWriter(this);

      // ... write the child text...
   }

   public void onChildElement(SAXElement element, SAXElement childElement,
                                    ExecutionContext executionContext)
                                       throws SmooksException,
IOException
   {

   }

   public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                throws SmooksException,
IOException
   {
      Writer writer = element.getWriter(this);
      // ... close the fragment...
   }
}
```

12.38. THE SAXELEMENT.SETWRITER

The *SAXElement.setWriter* lets you control the serialization of sub-fragments you need to reset the **`Writer`** instance so it diverts serialization of the sub-fragments.

Sometimes you know that the target fragment you are serializing/transforming will never have sub-fragments. In this situation, it is inefficient implement the **`SAXVisitBefore.visitBefore`** method just to make a call to the **`SAXElement.getWriter`** method to acquire ownership of the **`Writer`**. For this reason, we have the **`@StreamResultWriter`** annotation. Used in combination with the **`@TextConsumer`** annotation, it is only necessary to implement the **`SAXVisitAfter.visitAfter`** method.

12.39. STREAMRESULTWRITER EXAMPLE

```
@StreamResultWriter
```

```
public class MyVisitor implements SAXVisitAfter
{
   public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                    throws SmooksException,
IOException
   {
      Writer writer = element.getWriter(this);

      // ... serialize to the writer ...
   }
}
```

12.40. SAXTOXMLWRITER

Smooks provides the **`SAXToXMLWriter`** class. It simplifies the process of serializing **`SAXElement`** data as XML. This class allows you to write **`SAXVisitor`** implementations.

12.41. SAXTOXMLWRITER EXAMPLE

```
@StreamResultWriter
public class MyVisitor implements SAXElementVisitor
{
   private SAXToXMLWriter xmlWriter = new SAXToXMLWriter(this, true);

   public void visitBefore(SAXElement element, ExecutionContext
executionContext)
                                                 throws SmooksException,
IOException
   {
      xmlWriter.writeStartElement(element);
   }

   public void onChildText(SAXElement element, SAXText childText,
ExecutionContext
                               executionContext) throws SmooksException,
IOException
   {
      xmlWriter.writeText(childText, element);
   }

   public void onChildElement(SAXElement element, SAXElement childElement,
      ExecutionContext executionContext) throws SmooksException,
IOException
   {

   }

   public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                 throws SmooksException,
IOException
   {
```

```
      xmlWriter.writeEndElement(element);
   }
}
```

12.42. CONFIGURING THE SAXTOXMLWRITER

1. When writing a **SAXVisitor** implementation with the **SAXToXMLWriter**, set the **SAXToXMLWriter** constructor to a boolean. This is the **encodeSpecialChars** arg and it should be set based on the **rewriteEntities** filter setting.

2. Move the **@StreamResultWriter** annotation from the class and onto the **SAXToXMLWriter** instance declaration. This results in Smooks creating the **SAXToXMLWriter** instance which is then initialized with the **rewriteEntities** filter setting for the associated Smooks instance:

```
@TextConsumer
public class MyVisitor implements SAXVisitAfter
{
   @StreamResultWriter
   private SAXToXMLWriter xmlWriter;

   public void visitAfter(SAXElement element, ExecutionContext
executionContext)
                                                throws
SmooksException, IOException
   {
      xmlWriter.writeStartElement(element);
      xmlWriter.writeText(element);
      xmlWriter.writeEndElement(element);
   }
}
```

12.43. VISITOR CONFIGURATION

The **SAXVisitor** configuration is useful for testing purposes and works in exactly the same way as any other Smooks component. When configuring Smooks Visitor instances, the configuration **selector** is interpreted in a similar manner as an XPath expression. Visitor instances can be configured within program code on a Smooks instance.

12.44. EXAMPLE VISITOR CONFIGURATION

This example will use the **SAXVisitor** implementation as follows:

```
@TextConsumer
public class ChangeItemState implements SAXVisitAfter
{
   @StreamResultWriter
   private SAXToXMLWriter xmlWriter;

   @ConfigParam
   private String newState;

   public void visitAfter(SAXElement element, ExecutionContext
executionContext)
```

```
                                                    throws SmooksException,
IOException
   {
      element.setAttribute("state", newState);

      xmlWriter.writeStartElement(element);
      xmlWriter.writeText(element);
      xmlWriter.writeEndElement(element);
   }
}
```

Declaratively configuring **`ChangeItemState`** to fire on <order-item> fragments having a status of **`OK`** is shown below:

```
<smooks-resource-list
   xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

   <resource-config selector="order-items/order-item[@status = 'OK']">
      <resource>com.acme.ChangeItemState </resource>
      <param name="newState">COMPLETED</param>
   </resource-config>

</smooks-resource-list>
```

Custom configuration namespaces can be used to define a cleaner and more strongly typed configuration for the **`ChangeItemState`** component. A custom configuration namespace component is configured as follows:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
   xmlns:order="http://www.acme.com/schemas/smooks/order.xsd">

   <order:changeItemState itemElement="order-items/order-item[@status =
'OK']"
      newState="COMPLETED" />

</smooks-resource-list>
```

This Visitor could also be configured in source code as follows:

```
Smooks smooks = new Smooks();

smooks.addVisitor(new ChangeItemState().setNewState("COMPLETED"),
                                 "order-items/order-item[@status =
'OK']");

smooks.filterSource(new StreamSource(inReader), new
StreamResult(outWriter));
```

12.45. THE EXECUTIONLIFECYCLECLEANABLE

Visitor components implementing the **`ExecutionLifecycleCleanable`** life-cycle interface will be able to perform post **`Smooks.filterSource`** life-cycle operations. See the example below:

```
public interface ExecutionLifecycleCleanable extends Visitor
```

```
{
   public abstract void executeExecutionLifecycleCleanup(
                                          ExecutionContext
executionContext);
}
```

The basic call sequence can be described as follows (note the **`executeExecutionLifecycleCleanup`** calls):

```
smooks = new Smooks(..);

smooks.filterSource(...);
// ** VisitorXX.executeExecutionLifecycleCleanup **
smooks.filterSource(...);
// ** VisitorXX.executeExecutionLifecycleCleanup **
smooks.filterSource(...);
// ** VisitorXX.executeExecutionLifecycleCleanup **
 ... etc ...
```

This life-cycle method allows you to ensure that resources scoped around the **`Smooks.filterSource`** execution life-cycle can be cleaned up for the associated **`ExecutionContext`**.

12.46. THE VISITLIFECYCLECLEANABLE

Visitor components implementing the **`VisitLifecycleCleanable`** life-cycle interface will be able to perform post **`SAXVisitAfter.visitAfter`** life-cycle operations.

```
public interface VisitLifecycleCleanable extends Visitor
{
   public abstract void executeVisitLifecycleCleanup(ExecutionContext
executionContext);
}
```

The basic call sequence can be described as follows (note the **`executeVisitLifecycleCleanup`** calls):

```
smooks.filterSource(...);

    <message>
        <target-fragment>      < --- VisitorXX.visitBefore
            Text!!                      < --- VisitorXX.onChildText
            <child>                     < --- VisitorXX.onChildElement
            </child>
        </target-fragment>     < --- VisitorXX.visitAfter
        ** VisitorXX.executeVisitLifecycleCleanup **
        <target-fragment>      < --- VisitorXX.visitBefore
            Text!!                      < --- VisitorXX.onChildText
            <child>                     < --- VisitorXX.onChildElement
            </child>
        </target-fragment>     < --- VisitorXX.visitAfter
        ** VisitorXX.executeVisitLifecycleCleanup **
    </message>
    VisitorXX.executeExecutionLifecycleCleanup
```

```
smooks.filterSource(...);

    <message>
        <target-fragment>      < --- VisitorXX.visitBefore
            Text!!                     < --- VisitorXX.onChildText
            <child>                    < --- VisitorXX.onChildElement
            </child>
        </target-fragment>     < --- VisitorXX.visitAfter
        ** VisitorXX.executeVisitLifecycleCleanup **
        <target-fragment>      < --- VisitorXX.visitBefore
            Text!!                     < --- VisitorXX.onChildText
            <child>                    < --- VisitorXX.onChildElement
            </child>
        </target-fragment>     < --- VisitorXX.visitAfter
        ** VisitorXX.executeVisitLifecycleCleanup **
    </message>
    VisitorXX.executeExecutionLifecycleCleanup
```

This life-cycle method allows you to ensure that resources scoped around a single fragment execution of a **`SAXVisitor`** implementation can be cleaned up for the associated **`ExecutionContext`**.

12.47. THE EXECUTIONCONTEXT

The **`ExecutionContext`** is a context object for the storing of state information. It is scoped specifically around a single execution of a **`Smooks.filterSource`** method. All Smooks Visitor implementations must be stateless within the context of a single **`Smooks.filterSource`** execution, allowing the Visitor implementation to be used across multiple concurrent executions of the **`Smooks.filterSource`** method. All data stored in an **`ExecutionContext`** instance will be lost on completion of the **`Smooks.filterSource`** execution. The **`ExecutionContext`** is supplied in all Visitor API message event calls.

12.48. THE APPLICATIONCONTEXT

The **`ApplicationContext`** is a context object for storing of state information. It is scoped around the associated **`Smooks`** instance, that is, only one **`ApplicationContext`** instance exists per **`Smooks`** instance. This context object can be used to store data that needs to be maintained and accessible across multiple **`Smooks.filterSource`** executions. Components (including **`SAXVisitor`** components) can gain access to their associated **`ApplicationContext`** instance by declaring an **`ApplicationContext`** class property and annotating it with the **`@AppContext`** annotation. See the example below:

```
public class MySmooksComponent
{
    @AppContext
    private ApplicationContext appContext;

    // etc...
}
```

CHAPTER 13. ADVANCED

13.1. CONFIGURATION MODULARIZATION

NOTE

Since JBoss Fuse 6.3, the Smooks component for SwitchYard is deprecated and will be removed in a future release of JBoss Fuse.

Smooks configurations are modularized by the <import> element. This allows you to split Smooks configurations into multiple reusable configuration files and then compose the top level configurations using the <import>. You can also inject replacement tokens into the imported configuration by using <param> sub-elements on the <import>. This allows you to make tweaks to the imported configuration.

13.2. CONFIGURATION MODULARIZATION EXAMPLE

This is what the <import> element looks like in use.

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

    <import file="bindings/order-binding.xml" />
    <import file="templates/order-template.xml" />

</smooks-resource-list>
```

13.3. CONFIGURATION MODULARIZATION REPLACEMENT TOKEN EXAMPLE

In this example, replacement tokens have been injected. Th injection points are specified using @tokenname@:

```
<!-- Top level configuration... -->
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

    <import file="bindings/order-binding.xml">
        <param name="orderRootElement">order</param>
    </import>

</smooks-resource-list>

<!-- Imported parameterized bindings/order-binding.xml configuration... --
>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.4.xsd">

    <jb:bean beanId="order" class="org.acme.Order"
createOnElement="@orderRootElement@">
        .....
    </jb:bean>

</smooks-resource-list>
```

13.4. MULTI-RECORD FIELD DEFINITIONS

All flat file based reader configurations (including the CSV reader) support multi-record field definitions, which means that the reader can support CSV message streams that contain varying (multiple different types) CSV record types.

13.5. MULTI-RECORD FIELD DEFINITION EXAMPLE

This is a CSV message:

```
book,22 Britannia Road,Amanda Hodgkinson
magazine,Time,April,2011
magazine,Irish Garden,Jan,2011
book,The Finkler Question,Howard Jacobson
```

The CSV reader processes the message as shown:

```
<csv:reader fields="book[name,author] | magazine[*]"
rootElementName="sales" indent="true" />
```

This is the resulting output:

```
<sales>
    <book number="1">
        <name>22 Britannia Road</name>
        <author>Amanda Hodgkinson</author>
    </book>
    <magazine number="2">
        <field_0>Time</field_0>
        <field_1>April</field_1>
        <field_2>2011</field_2>
    </magazine>
    <magazine number="3">
        <field_0>Irish Garden</field_0>
        <field_1>Jan</field_1>
        <field_2>2011</field_2>
    </magazine>
    <book number="4">
        <name>The Finkler Question</name>
        <author>Howard Jacobson</author>
    </book>
</sales>
```

Note the syntax in the 'fields' attribute. Each record definition is separated by the pipe character '|'. Each record definition is constructed as record-name[field-name,field-name]. record-name is matched against the first field in the incoming message and so used to select the appropriate record definition to be used for outputting that record. Also note how you can use an asterisk character ('*') when you do not want to name the record fields. In this case (as when extra/unexpected fields are present in a record), the reader will generate the output field elements using a generated element name e.g. "field_0", "field_1" etc.

13.6. WILDCARD BINDINGS

Virtual models support "wildcard" <jb:value> bindings. That is, you can bind all the child elements of an element into a Map using a single configuration, where the child element names act as the Map entry key and the child element text value acts as the Map entry value. To do this, you simply omit the property attribute from the the <jb:value> configuration and use a wildcard in the data attribute.

13.7. WILDCARD BINDING EXAMPLE

In the following example, a <order-item> element has been set. It contains values to be populated into a Map:

```
<order-item>
    <product>111</product>
    <quantity>2</quantity>
    <price>8.90</price>
</order-item>
```

The wildcard binding configuration for doing this would be:

```
<jb:bean beanId="orderItem" class="java.util.HashMap"
createOnElement="order-items/orderItem">
    <jb:value data="order-items/orderItem/*" />
</jb:bean>
```

13.8. THE XMLBINDING CLASS

The XMLBinding class is a special utility wrapper class around the Smooks runtime. It is designed specifically for reading and writing XML data to and from Java Object models using nothing more than standard <jb:bean> configurations. It allows users to leverage frameworks like JAXB or JiBX (read and write between Java and XML using a single configuration), but with the added advantage of being able to easily handle multiple versions of an XML schema/model in a single Java model. Users can read and write multiple versions of an XML message into a single Java object model. Messages can be transformed from one version to another by reading the XML into the common Java object model using an XMLBinding instance configured for one version of the XML, and then writing those Java objects back out using an XMLBinding instance configured for the other version of the XML.

13.9. XMLBINDING CLASS EXAMPLE

This is what the XMLBinding class looks like in a configuration:

```
// Create and initilize the XMLBinding instance...
XMLBinding xmlBinding = new XMLBinding().add("/smooks-configs/order-xml-
binding.xml");
xmlBinding.intiailize();

// Read the order XML into the Order Object model...
Order order = xmlBinding.fromXML(new StreamSource(inputReader),
Order.class);

// Do something with the order....

// Write the Order object model instance back out to XML...
xmlBinding.toXML(order, outputWriter);
```

13.10. TRANSFORMING XML MESSAGES USING XMLBINDING EXAMPLE

This is how the XMLBinding class is used to transform messages:

```
// Create and initilise the XMLBinding instances for v1 and v2 of the
XMLs...
XMLBinding xmlBindingV1 = new XMLBinding().add("v1-binding-config.xml");
XMLBinding xmlBindingV2 = new XMLBinding().add("v2-binding-config.xml");
xmlBindingV1.intiailize();
xmlBindingV2.intiailize();

// Read the v1 order XML into the Order Object model...
Order order = xmlBindingV1.fromXML(new StreamSource(inputReader),
Order.class);

// Write the Order object model instance back out to XML using the v2
XMLBinding instance...
xmlBindingV2.toXML(order, outputWriter);
```

13.11. MAP CREATION FROM RECORD SET EXAMPLE

Smooks supports the creation of Maps from a record set:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:ff="http://www.milyn.org/xsd/smooks/flatfile-1.5.xsd">

    <ff:reader fields="firstname,lastname,gender,age,country"
parserFactory="com.acme.PersonRecordParserFactory">
        <ff:mapBinding beanId="people" class="com.acme.Person"
keyField="firstname" />
    </ff:reader>

</smooks-resource-list>
```

13.12. MAP CREATION FROM RECORD SET EXECUTION EXAMPLE

A map created from a record set would be executed as follows:

```
Smooks smooks = new Smooks(configStream);
JavaResult result = new JavaResult();

smooks.filterSource(new StreamSource(messageReader), result);

Map<String, Person> people = (Map<String, Person>)
result.getBean("people");

Person tom = people.get("Tom");
Person mike = people.get("Mike");
```

13.13. VARIABLEFIELDRECORDPARSER AND VARIABLEFIELDRECORDPARSERFACTORY

VariableFieldRecordParser and VariableFieldRecordParserFactory are abstract implementations of the RecordParser and RecordParserFactory interface. They provide very useful base implementations for a Flat File Reader, providing base support for:

- Utility Java binding configurations
- Support for "variable field" records, that is, a flat file message that contains multiple record definitions
- The ability to read the next record chunk, with support for a simple record delimiter, or a regular expression (Regex) pattern that marks the beginning of each record
- The CSV and Regex readers are implemented using these abstract classes

13.14. RECORD DEFINITION EXAMPLE

This is what a record definition looks like:

```
fields="book[name,author] | magazine[*]
```

- The record definitions are pipe separated: "book" records will have a first field value of "book" while "magazine" records will have a first field value of "magazine"
- Asterisk ("*") as the field definition for a record basically tells the reader to generate the field names in the generated events (e.g. "field_0", "field_1" etc.)