V1.2:Smooks v1.2 User Guide

From Smooks

Contents

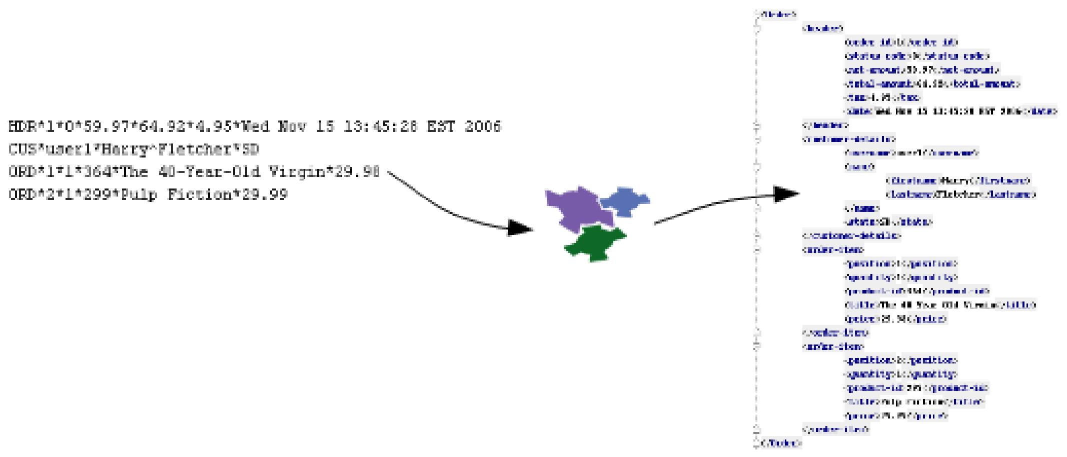
Overview

Smooks is a Java Framework/Engine for processing **XML** and **non XML** data (CSV, EDI, Java etc).

The following are some of the features available in **Smooks v1.2**:

- **Transformation**

  Perform a wide range of **Data Transforms** - XML to XML, CSV to XML, EDI to XML, XML to EDI, XML to CSV, Java to XML, Java to EDI, Java to CSV, **Java to Java**, XML to Java, EDI to Java etc.

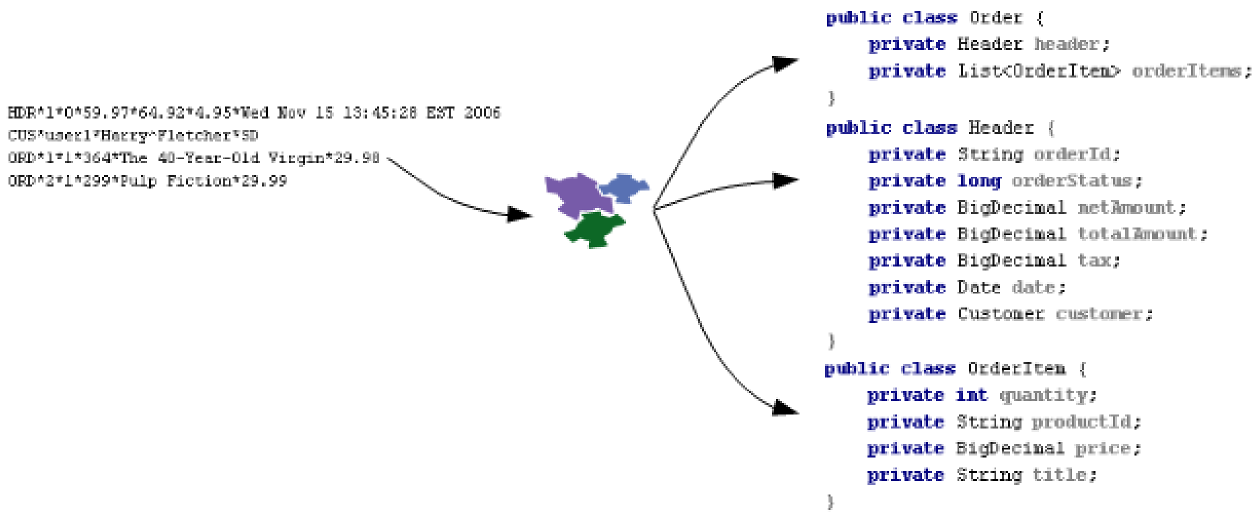


```
HDR*1*0*59.97*64.92*4.95*Wed Nov 15 13:45:28 EST 2006
CUS*user1*Harry^Fletcher*SD
ORD*1*1*364*The 40-Year-Old Virgin*29.98
ORD*2*1*299*Pulp Fiction*29.99
```

```
<Order>
    <header>
        <order-id>1</order-id>
        <status-code>0</status-code>
        <net-amount>59.97</net-amount>
        <total-amount>64.92</total-amount>
        <tax>4.95</tax>
        <date>Wed Nov 15 13:45:28 EST 2006</date>
    </header>
    <customer-details>
        <username>user1</username>
        <name>
            <firstname>Harry</firstname>
            <lastname>Fletcher</lastname>
        </name>
        <state>SD</state>
    </customer-details>
    <order-item>
        <position>1</position>
        <quantity>1</quantity>
        <product-id>364</product-id>
        <title>The 40 Year Old Virgin</title>
        <price>29.98</price>
    </order-item>
    <order-item>
        <position>2</position>
        <quantity>1</quantity>
        <product-id>299</product-id>
        <title>Pulp Fiction</title>
        <price>29.99</price>
    </order-item>
</Order>
```

- **Java Binding**

  Populate a Java Object Model from a data source (CSV, EDI, XML, Java etc). Populated object models can be used as a transformation result itself, or can be used by (e.g.) Templating resources for generating XML or other character based results. Also supports **Virtual Object Models** (Maps and Lists of typed data), which can be used by EL and Templating functionality.

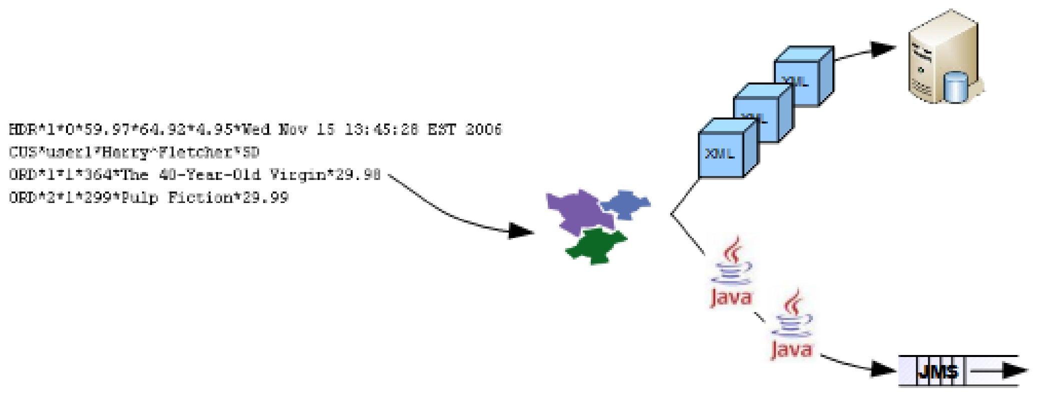


```
HDR*1*0*59.97*64.92*4.95*Wed Nov 15 13:45:28 EST 2006
CUS*user1*Harry^Fletcher*SD
ORD*1*1*364*The 40-Year-Old Virgin*29.98
ORD*2*1*299*Pulp Fiction*29.99
```

```
public class Order {
    private Header header;
    private List<OrderItem> orderItems;
}
public class Header {
    private String orderId;
    private long orderStatus;
    private BigDecimal netAmount;
    private BigDecimal totalAmount;
    private BigDecimal tax;
    private Date date;
    private Customer customer;
}
public class OrderItem {
    private int quantity;
    private String productId;
    private BigDecimal price;
    private String title;
}
```

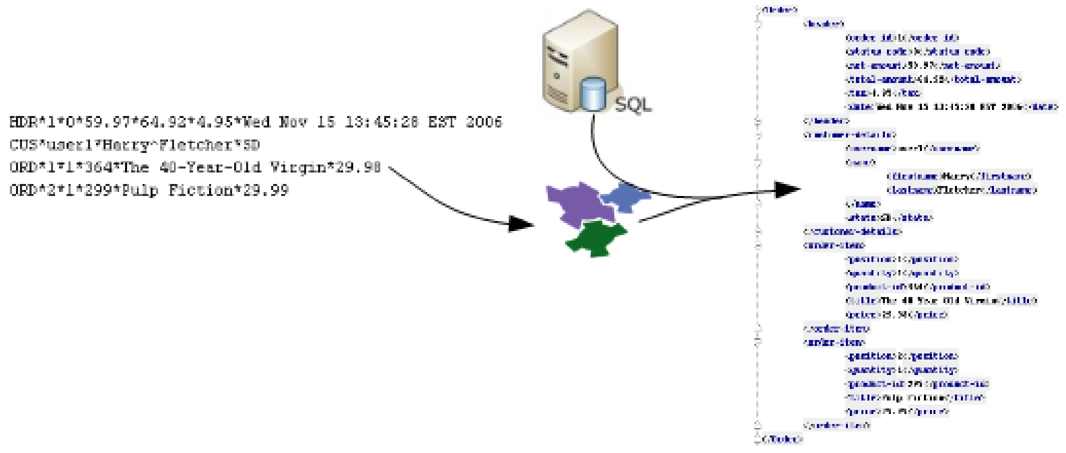
- **Huge Message Processing**

  Process **huge messages** (GBs) - **Split**, **Transform** and **Route** message fragments to **JMS**, **File**, **Database** etc destinations.


HDR*1*0*59.97*64.92*4.95*Wed Nov 15 13:45:28 EST 2006
CUS*user1*Harry^Fletcher*SD
ORD*1*1*364*The 40-Year-Old Virgin*29.98
ORD*2*1*299*Pulp Fiction*29.99
XML
XML
XML
Java
Java
JMS

- **Message Enrichment**

  Enrich a message with data from a **Database**, or other Datasources.


HDR*1*0*59.97*64.92*4.95*Wed Nov 15 13:45:28 EST 2006
CUS*user1*Harry^Fletcher*SD
ORD*1*1*364*The 40-Year-Old Virgin*29.98
ORD*2*1*299*Pulp Fiction*29.99
SQL

- **Complex Message Validation**

  Rules based fragment validation.

- **ORM Based Message Persistence**

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

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

- **Combine**

  Perform **Extract Transform Load** (ETL) operations by leveraging Smooks' **Transformation**, **Routing** and **Persistence** functionality.

Getting Started

The easiest way to get started with Smooks is to download and try out some of the examples. The examples are the perfect base upon which to integrate Smooks into your application.

FAQs

See the FAQ

Maven

For details on how to integrate Smooks into your project via Maven, see the Maven & Ant Guide.

Ant

For details on how to integrate Smooks into your project via Ant, see the Maven & Ant Guide.

Basics

The most commonly accepted definition of Smooks would be that it is a "Transformation Engine". However, at it's core, Smooks makes no mention of "data transformation". The **smooks-core** codebase is designed simply to support hooking of custom "Visitor" logic into an Event Stream produced by a data Source of some kind (XML, CSV, EDI, Java etc). As such, **smooks-core** is simply a "**Structured Data Event Stream Processor**".

Of course, the most common application of this will be in the creation of **Transformation** solutions i.e. implementing Visitor logic that uses the Event Stream produced from a Source message to produce a Result of some other kind. The capabilities in **smooks-core** enable more than this however. We have implemented a range of other solutions based on this processing model:

- **Java Binding**: Population of a Java Object Model from the Source message.
- **Message Splitting & Routing**: The ability to perform complex splitting and routing operations on the Source message, including routing to multiple destinations concurrently, as well as routing different data formats concurrently (XML, EDI, CSV, Java etc).
- **Huge Message Processing**: The ability to declaratively consume (transform, or split and route) huge message without writing lots of high maintenance code.

Basic Processing Model

As stated above, the basic principal of Smooks is to take a data **Source** of some kind (e.g. XML) and from it generate an **Event Stream**, to which you apply **Visitor logic** to produce a **Result** of some other kind (e.g. EDI).

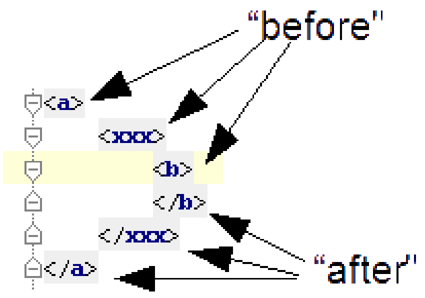
Many different data Source and Result types are supported, meaning many different transformation types are supported, including (but not limited to):

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

- Java to EDI
- CSV to XML
- CSV to ...
- etc etc

In terms of the Event Model used to map between the Source and Result, Smooks currently supports DOM and SAX Event Models. We will concentrate on the SAX event model here. If you want low level details on either models, please consult the Smooks Developer Guide (http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_Developer_Guide) . The SAX event model is based on the hierarchical SAX events generated from an XML Source (startElement, endElement etc). However, this event model can be just as easily applied to other structured/hierarchical data Sources (EDI, CSV, Java etc).

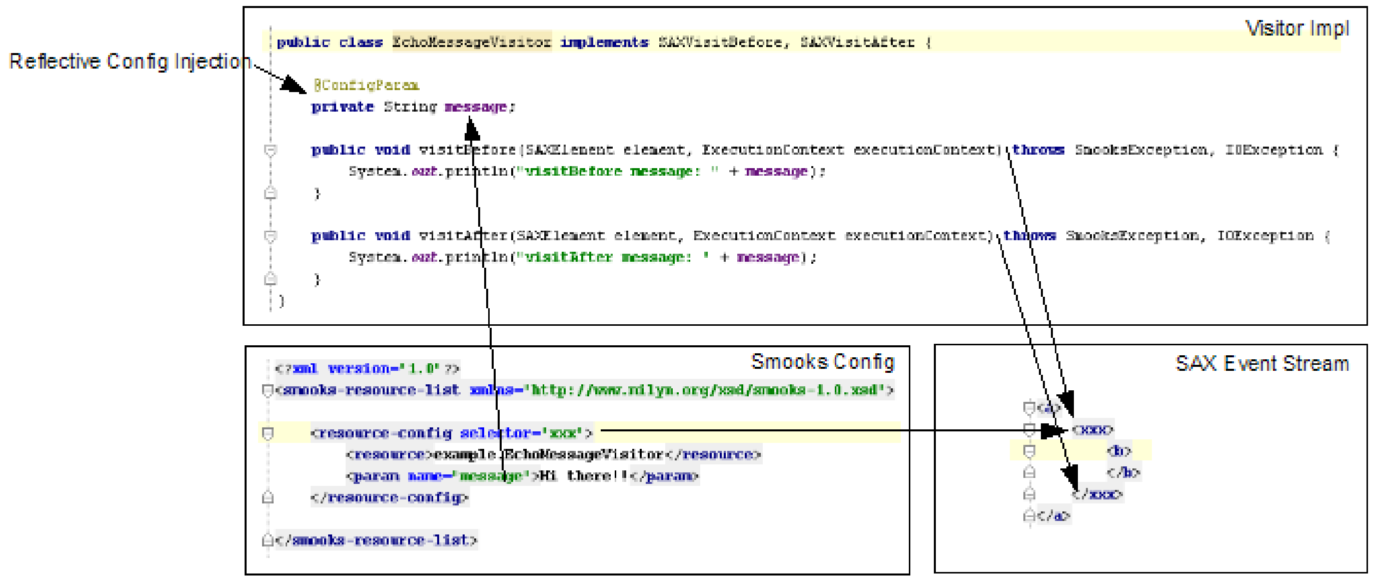
The most important events (typically) are the **visitBefore** and **visitAfter** events. The following illustration tries to convey the hierarchical nature of these events.



Simple Example

In order to consume the SAX Event Stream produced from the Source message, you need to implement one or more of the SAXVisitor (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/delivery/sax/SAXVisitor.html) interfaces (depending on which events you need to consume).

The following is a very simple example of how you implement Visitor logic and target that logic at the **visitBefore** and **visitAfter** events for a specific element in the Event Stream. In this case we target the Visitor logic at the <xxx> element events.



```
public class EchoMessageVisitor implements SAXVisitBefore, SAXVisitAfter {

    @ConfigParam
    private String message;

    public void visitBefore(SAXElement element, ExecutionContext executionContext) throws SmooksException, IOException {
        System.out.println("visitBefore message: " + message);
    }

    public void visitAfter(SAXElement element, ExecutionContext executionContext) throws SmooksException, IOException {
        System.out.println("visitAfter message: " + message);
    }
}
```

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

    <resource-config selector='xxx'>
        <resource>example.EchoMessageVisitor</resource>
        <param name='message'>Hi there!!</param>
    </resource-config>

</smooks-resource-list>
```

As you can see, the Visitor implementation is very simple; one method implementation per event. To target this implementation at the <xxx> element **visitBefore** and **visitAfter** events, we need to create a Smook configuration as shown (more on "Resource Configurations" in the following sections).

The Smooks code to execute this is very simple:

```
Smooks smooks = new Smooks("/smooks/echo-example.xml");

smooks.filterSource(new StreamSource(inputStream));
```

Note that in this case we don't produce a Result. Also note that we don't interact with the "execution" of the filtering process in any way, since we don't explicitly create an ExecutionContext (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/container/ExecutionContext.html) and supply it to the Smooks (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/Smooks.html) .filterSource method call.

This example illustrated the lower level mechanics of the Smooks Programming Model. In reality however, users are not going to want to solve their problems by implementing lots Java code themselves from scratch. For this reason, Smooks is shipped with quite a lot of pre-built functionality i.e. ready to use Visitor logic. We bundle this Visitor logic based on functionality and we call the bundles "**Cartridges**".

Smooks Cartridges

The basic functionality of Smooks Core can be extended through the creation of what we call a "Smooks Cartridge". A Cartridge is simply a Java archive (jar) containing reusable Content Handlers (Visitor Logic). A Smooks Cartridge should provide "ready to use" support for a specific type of XML analysis or transformation.

For a full list of the Cartridges supported by Smooks, see the Cartridges list (http://www.smooks.org/cartri) .

Filtering Process Selection (DOM or SAX?)

This is done by Smooks based on the following criteria:

1. If all Visitor resources* implement only the DOM Visitor interfaces (DOMElementVisitor or SerializationUnit), then the DOM processing model is selected.
2. If all Visitor resources* implement only the SAX Visitor interface (SAXElementVisitor), then the SAX processing model is selected.
3. If all Visitor resources* implement both the DOM and SAX Visitor interfaces, then the DOM processing model is selected, unless the Smooks resource configuration contains the **stream.filter.type** global configuration parameter (see below).

* not including non element Visitor resources, like readers for example.

The **stream.filter.type** can be set to either DOM or SAX. The following is an examlpe of setting the stream.filter.type to SAX:

```
<params>
    <param name="stream.filter.type">SAX</param>
</params>
```

For more information about global parameters can be found in the section Global Configurations (http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Global_Configurations) .

Mixing DOM and SAX

The DOM processing model has the obvious:

- **Advantage** of being easier to work with on a code level, allowing node traversal etc. It also makes it a lot easier to take advantage of Scripting and Templating engines that have built in support for utilizing DOM structures (e.g. FreeMarker (http://freemarker.org) and Groovy (http://groovy.codehaus.org)).
- **Disadvantage** of being constrained by memory i.e. if you have huge messages, then you typically cannot use a DOM processing model.

Smooks v1.1 added support for mixing these 2 models through the **DomModelCreator** class. When used with SAX filtering, this Visitor will construct a DOM Fragment of the visited element. This allows DOM utilities to be used in a Streaming environment.

When 1+ models are nested inside each other, outer models will never contain data from the inner models i.e. the same fragments will never coexist inside two models.

Take the following message as an example:

```
<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>
```

The DomModelCreator can be configured in Smooks to create models for the "order" and "order-item" message fragments:

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

In this case, the "order" model will never contain "order-item" model data (order-item elements are nested inside the order element). The in memory model for the "order" will simply be:

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

Added to this is the fact that there will only ever be 0 or 1 "order-item" models in memory at any given time, with each new "order-item" model overwriting the previous "order-item" model. **All this ensures that the memory footprint is kept to a minimum**.

Because the Smooks processing model is event driven via the message content (i.e. you can hook in Visitor logic to be applied at different points while Smooks filters/streams the message), you can take advantage of this mixed DOM and SAX processing model.

See the following examples that utilize this mixed DOM + SAX approach:

- Groovy Scripting Example
- FreeMarker Templating Example

Checking the Smooks Execution Process

As Smooks performs the filtering process (processing the Event Stream generated from the Source), it publishes events that can be captured and programmatically analyzed during/after execution.? The easiest way to generate an execution report out of Smooks is to configure the ExecutionContext to generate a report. Smooks supports generation of a HTML report via the HtmlReportGenerator class.

The following is an example of 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));
```

The HtmlReportGenerator is a very useful tool during development with Smooks. It's the nearest thing Smooks currently has to an IDE based Debugger (which we hope to have in a future release). It can be very useful for diagnosing issues, or simply as a tool for comprehending a Smooks transformation.

An example HtmlReportGenerator report can be seen online here (http://milyn.codehaus.org/smooks-report /report.html) .

Of course you can also write and use your own ExecutionEventListener (http://www.milyn.org/javadoc /v1.2/smooks/org/milyn/event/ExecutionEventListener.html) implementations.

Smooks Resources

Smooks executes by taking a data stream of one form or another (XML, EDI, Java, JSON, CSV etc) and from it, it generates an event stream, which it uses to fire different types of "Visitor logic" (Java, Groovy, FreeMarker, XSLT etc). The result of this process can be to produce a new data stream in a different format (i.e. a traditional "transformation"), bind data from the source message data stream to java objects to produce a populated Java object graph (i.e. a "Java binding"), produce many smaller messages (message splitting) etc.

At a core level (inside Smooks), it just sees all of the "Visitor logic" etc as "Smooks Resources" (SmooksResourceConfiguration) that are configured to be applied based on an event **selector** (i.e. event from the source data event stream). This is a very generic processing model and makes a lot of sense from the point of view of Smooks Core and it's architecture (maintainance etc). However, it can be a little too generic from a usability perspective because everything looks very similar in the configuration. To help with this, Smooks v1.1 introduced an "Extensible Configuration Model" feature. This allows specific resource types (Javabean binding configs, FreeMarker template configs etc) to be specified in the configuration using dedicated XSD namespaces of their own.

Example (Java Binding Resource):

```
<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>
```

Example (FreeMarker Template Resource):

```
<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>
```

When comparing the above examples to the pre Smooks v1.1 equivalents you can see that:

1. The user now has a more strongly typed configuration that is domain specific in each case (and so easier to read).
2. Because the v1.1+ configurations are XSD based, the user also gets autocompletion support from their IDE.
3. No longer any need to define the actual handler for the given resource type e.g. the BeanPopulator for java bindings.

Java Binding

The Smooks JavaBean Cartridge allows you to create and populate Java objects from your message data (i.e. bind data to).

Java Binding Overview

This feature of Smooks can be used in its own right purely as a Java binding framework for XML, EDI, CSV etc. However, it is very important to remember that the Java Binding capabilities in Smooks are the cornerstone of many other capabilities provided by Smooks. This is because Smooks makes the Java Objects it creates (and binds data into) available through the BeanRepository (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/javabean/org/milyn/javabean/repository/BeanRepository.html) class. This is essentially a java bean context that is made available to any Smooks Visitor implementation via the Smooks ExecutionContext (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/container/ExecutionContext.html) .

Some of the existing features that build on the functionality provided in the Javabean Cartridge include:

- Templating: Templating typically involves applying a template (FreeMarker or other) to the objects in the BeanRepository.
- Validation: Business Rules Validation (e.g. via MVEL) typically involves applying a rule (expression etc) to the objects in the BeanRepository.
- Message Splitting & Routing: Message Splitting typically works by generating split messages from the Objects in the BeanRepository, either by using the objects themselved and routing them, or by applying a template to them and routing the result of that templating operation (e.g. a new XML, CSV etc).
- Persistence (Database Reading and Writing): The Persistence features depend on the Java Binding functions for creating and populating the Java Objects (Entities etc) to be persisted. Data read from a database is typically bound into the BeanRepository.
- Message Enrichment: As stated above, enrichment data (e.g. read from a DB) is typically bound into the BeanRepository, from where it is available to all other features, including the Java Binding functionality itself e.g. for expression based bindings. This allows messages generated by Smooks to be enriched.

When to use Smooks Java Binding

A question that often comes to peoples minds is "*Why would I use Smooks (instead of JAXB/JiBX (http://jibx.sourceforge.net/) etc) to perform binding to a Java Objects model?*". Well there are a number of reasons why you would use Smooks and there are a number of reasons why you would not use Smooks.

When Smooks makes sense:

1. Binding non-XML data to a Java Object model e.g. EDI, CSV, JSON etc
2. Binding data (XML or other) whose data model (hierarchical structure) does not match that of the target Java Object model. JiBX (http://jibx.sourceforge.net/) also supports this, but only for XML (AFAIK!!).
3. When you are binding data from an XML data structure for which there is no defined schema (XSD). Some frameworks effectively require a well defined XML data model via schema.
4. When binding data from multiple existing and different data formats into a single pre-existing Java Object model. Related to the above points.
5. When binding data into existing 3rd Party Object Models that you cannot modify e.g. through a post-compile step.
6. In situations where the Data (XML or other) and Java Object models may vary in isolation from each other. Because of #2 above, Smooks can handle this by simply modifying the binding configuration. Other frameworks often require binding/schema regeneration, redeployment etc (see #3 above).
7. Where you need to execute additional logic in parallel to the binding process e.g. Validation, Split Message Generation (via Templates), Split Message Routing, Fragment Persistence, or any custom logic that you may wish to implement. This is often a very powerful capability e.g. when processing huge message streams.
8. Processing huge message streams by splitting them into a series of many small Object models and routing them to other systems for processing.
9. When using other Smooks features that rely on the Smooks Java Binding capabilities.

When Smooks may not make sense:

1. When you have a well defined data model (via schema/XSD) and all you need to do is bind data into an Object model (no validation, persistence etc required).
2. When the object model is isolated from other systems and so can change without impacting such systems.
3. Where processing XML and performance is paramount over all other considerations (where nanoseconds matter), frameworks such as JiBX (http://jibx.sourceforge.net/) are definitely worth considering over Smooks. This is not to imply that the performance of Smooks Java Binding is poor in any way, but it does acknowledge the fact that frameworks that utilise post-compile optimizations targeted at a specific data format (e.g. XML) will always have the edge under the right conditions.

Basics Of Java Binding

As you know, Smooks supports a range of source data formats (XML, EDI, CSV, Java etc), but for the purposes of this topic, we will always refer to the message data in terms of an XML format. In the examples, we will continuously refer to the following XML message:

```
<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>
```

In some examples we will use different XML message data. Where this happens, the data is explicitly defined there then.

The JavaBean Cartridge is used via the http://www.milyn.org/xsd/smooks/javabean-1.2.xsd (http://www.milyn.org/xsd/smooks/javabean-1.2.xsd) configuration namespace. Install the schema in your IDE and avail of autocompletion.

An example configuration:

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

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

</smooks-resource-list>
```

This configuration simply creates an instance of the example.model.Order class and binds it into the **bean context** under the beanId "order". The instance is created at the very start of the message on the #document element (i.e. the start of the root <order> element).

- **beanId**: The id of this bean. Please see The Bean Context for more details.
- **class**: The fully qualified class name of the bean.
- **createOnElement**: attribute controls when the bean instance is created. Population of the bean properties is controlled through the binding configurations (child elements of the <jb:bean> element).
- **createOnElementNS**: The namespace of the createOnElement can be specified via the **createOnElementNS** attribute.

The Bean Context

The **bean context** (also known as "bean map") is a very important part of the JavaBean Cartridge. One bean context is created per execution context (i.e. per Smooks.filterSource operation). Every bean, created by the cartridge, is put into this context under its **beanId**. If you want the contents of the bean context to be returned at the end of the Smooks.filterSource process, supply a **org.milyn.delivery.java.JavaResult** object in the call to Smooks.filterSource method. The following example illustrates this principal:

```
//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");
```

If you need to access the bean context beans at runtime (e.g. from a customer Visitor implementation), you do so via the **BeanRepository** class.

The Javabean cartridge has the following conditions for javabeans:

1. A public no-argument constructor
2. Public property setter methods. The don't need to follow any specific name formats, but it would be better if they do follow the standard property setter method names.
3. Setting javabean properties directly is not supported.

Java Binding Configuration Details

The configuration shown above simply created the *example.model.Order* bean instance and bound it into the bean context. This section will describe how to bind data into that bean instance.

The Javabean Cartridge provides support for 3 types of data bindings, which are added as child elements of the **<jb:bean>** element:

- **<jb:value>**: This is used to bind data values from the Source message event stream into the target bean.
- **<jb:wiring>**: This is used to "wire" another bean instance from the bean context into a bean property on the target bean. This is the configuration that allows you to construct an object graph (Vs just a loose bag of Java object instances).
- **<jb:expression>**: As it's name suggests, this configuration is used to bind in a value calculated from an

expression, a simple example being the binding of an order item total value into an OrderItem bean based on the result of an expression that calculates the value from the items price and quantity (e.g. "price * quantity").

Taking the Order XML message (previous section), lets see what the full XML to Java binding configuration might be. We've seen the order XML (above). Now lets look at the Java Objects that we want to populate from that XML message (getters and setters 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;
}
```

The Smooks config required to bind the data from the order XML and into this object model 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

(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" >
               header.total + (orderItem.price * orderItem.quantity);
           </jb:expression>
       </jb:bean>

(3)    <jb:bean beanId="orderItems" class="java.util.ArrayList" createOnElement="order">
(3.a)      <jb:wiring 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>
```

(1)	Configuration **(1)** defines the creation rules for the *com.acme.Order* bean instance (top level

	bean). We create this bean instance at the very start of the message i.e. on the <order> element . In fact, we create each of the beans instances (**(1)**, **(2)**, **(3)** - all accepts **(4)**) at the very start of the message (on the <order> element). We 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 *List<OrderItem>* 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.
(2)	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 Date Decoder (http://www.milyn.org/javadoc/v1.2/commons/org/milyn/javabean/decoders/DateDecoder.html) . Configuration **(2.b)** defines a **value** binding configuration onto *Header.customerNumber* property. What should be noted here is 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).
(3)	Configuration **(3)** creates the *List<OrderItem>* bean instance for holding the *OrderItem* instances. Configuration **(3.a)** wires in the orderItem bean (**(4)**) instances 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).
(4)	Configuration **(4)** creates the *OrderItem* bean instances. Note how the **createOnElement** is set to the <order-item> element. This is because we want a new instance of this bean to be created for every <order-item> element (and wired into the *List<OrderItem>* **(3.a)**). If the **createOnElement** attribute for this configuration was not set to the <order-item> element (e.g. if 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 i.e. you would be left with a *List<OrderItem>* with just a single *OrderItem* instance containing the <order-item> data from the last <order-item> encountered in the source message.

Binding Tips

- **<jb:bean createOnElement>**
  1. Set it to the root element (or "#document"): For bean instances where only a single instance will exist in the model.
  2. Set it to the recurring element: For Collection bean instances. If you don't specify the correct element in this case, you could loose data.
- **<jb:value decoder>**
  1. In most cases, Smooks will automatically detect the datatype decoder to be used for a <jb:value> binding. However, some decoders require configuration e.g. the Date decoder (decoder="Date"). In these cases, the decoder attribute should be defined on the binding, as well as the <jb:decodeParam> child elements for specifying the decode parameters for that decoder. See the full list of DataDecoder available out of the box (http://www.milyn.org/javadoc/v1.2/commons/org/milyn/javabean/decoders/package-summary.html) .
- **<jb:wiring property>**
  1. Not required when binding into Collections.
- **<jb:expression execOnElement>**
  1. Explicitly tells Smooks when the expression is to be evaluated and the result bound. If not defined, the expression is executed based on the value of the parent <jb:bean createOnElement>.
- **Collections**
  1. Just define the <jb:bean class> to be the required Collection type and wire in the Collection entries.
  2. For arrays, just postfix the <jb:bean class> attribute value with square brackets e.g. **class="com.acme.OrderItem[]"**.

Extended Lifecycle Bindings

Binding Key Value Pairs into Maps

If the <jb:value property> attribute of a binding is not defined (or is empty), then the name of the selected node will be used as the map entry key (where the beanClass is a Map).

There is one other way to define the map key. The value of the <jb:value property> attribute can start with the **@** character. The rest of the value then defines the attribute name of the selected node, from which the map key is selected. The following example demonstrates this:

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

An 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**].

Of course 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.

Virtual Object Models (Maps & Lists)

It is possible to create a complete object model without writing your own Bean classes. This virtual model is created using only maps and lists . This is very convenient if you use the javabean cartridge between two processing steps. For example from xml -> java -> edi.

The following example demonstrates the principle:

```
<?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

    <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>

</smooks-resource-list>
```

Note above how we always define the **decoder** attribute for a Virtual Model (Map). This is because Smooks has no way of auto-detecting the decode type for the data binding to a Map. So, if you need types 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.

Take a look at the **milyn/smooks-examples/xml-to-java-virtual** for another example.

Merging Multiple Data Entities Into a Single Binding

This can be achieved using Expression Based Bindings (<jb:expression>).

Generating the Smooks Binding Configuration

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.

From the commandline:

```
$JAVA_HOME/bin/java -classpath <classpath> org.milyn.javabean.gen.ConfigGenerator -c <rootBeanClass> -o <output
```

- 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 aditional 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.

After running this utility against the target class, you typically need to perform the following follow-up tasks in order 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 bean 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 e.g. you may need to configure **<jb:decodeParam>** sub-elements for the decoder on some of the bindings. E.g. for a date field.
4. Double-check the binding config elements (**<jb:value>** and **<jb:wiring>**), making sure all Java properties have been covered in the generated configuration.

Determining the selector values can sometimes be difficult, especially for non XML Sources (Java etc). The Html Reporting tool can be a great help here because it helps you visualise the input message model (against which the selectors will be applied) as seen by Smooks. So, first off, 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.

The following is an example of a generated configuration. 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

    <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>
```

Programmatic Configuration

Java Binding Configuratons can be programmatically added to a Smooks using the Bean (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/javabean/org/milyn/javabean/Bean.html) 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.

Example

Taking the "classic" Order message as an example and binding it into a corresponding Java Object model.

The Message:

```
<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");
```

Notes on JavaResult

Users should note that there is **no guarantee** as to the exact contents of a JavaResult (http://www.milyn.org /javadoc/v1.2/smooks/org/milyn/payload/JavaResult.html) 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.

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-1.2.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>
```

So after applying this configuration, calls to the JavaResult.getBean(String) method for anything other than the "order" bean will return null. This will work fine in cases such as the above example, because the other bean instances are wired into the "order" graph.

Note that as of Smooks v1.2, if a JavaSource (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/payload /JavaSource.html) 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 (http://www.milyn.org /javadoc/v1.2/smooks/org/milyn/container/ExecutionContext.html) for that Smooks.filterSource invocation. This will mean that some of the JavaSource bean instances may be visible in the JavaResult.

Templating

Smooks provides two main Templating options:

1. FreeMarker Templating (http://freemarker.org)
2. XSL Templating (http://www.w3.org/Style/XSL/)

What Smooks adds here is the ability to use these Templating technologies within the context of a Smooks filtering process. This means that these technologies:

1. Can be applied to a source message on a per fragment basis Vs the whole message i.e. "Fragment Based Transforms". This is useful in situations where (for example) you only wish to insert a piece of data into a message at a specific point (e.g. add headers to a SOAP message), but you don't wish to interfere with the rest of the message stream. In this case you can "target" (apply) the template to the fragment of interest.
2. Can take advantage of other Smooks technologies (Cartridges) such as the Javabean Cartridge. In this scenario, you can use the Javabean Cartridge to decode and bind data from the message into the Smooks bean context and then use (reference) that decoded data from inside your FreeMarker template (Smooks makes this data available to FreeMarker).
3. Can be used to process huge message streams (GBs), while at the same time maintain a relatively simple processing model, with a low memory footprint. See Processing Huge Messages (http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Processing_Huge_Messages_.28GBs.29) .
4. Can be used for generating "Split Message Fragments" that can then be routed (using Smooks Routing components (http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing)) to physical endpoints (File, JMS), or logical endpoints on an ESB (a "Service").

Smooks can also be extended (and will) to add support for other templating technologies.

Note: *Be sure to read the section on Java Binding.*

FreeMarker Templating

FreeMarker (http://freemarker.org) is a very powerful Templating Engine. Smooks allows FreeMarker to be used as a means of generating text based content that can then be inserted into a message stream (aka a "Fragment Transform"), or used as a "Split Message Fragment" for routing to another process (http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing) .

Configuring FreeMarker templates in Smooks is done through the http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd configuration namespace. Just configure this XSD into your IDE and you're in business!

Example - Inline Template:

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

Example - External Template Reference:

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

Smooks allows you to perform a number of operations with the Templating result. This is controlled by the **<use>** element, which is added to the <ftl:freemarker> configuration.

Example - Inlining the Templating Result:

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

Inlining allows you to inline the templating result into a **Smooks.filterSource** Result object. A number of **directives** are supported:

- **addto**: Add the templating result to the targeted element.
- **replace** (default): Use 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**: Add the templating result before to the targeted element.
- **insertafter**: Add the templating result after to the targeted element.

Using **<ftl:bindTo>**, you can bind the Templating result to the Smooks **bean context**. The templating result can then be accessed by other Smooks components, such as the routing components. This can be especially useful for splitting (http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing) huge messages into smaller (more consumable) messages that can then be routed (http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing) to another process for handling.

Example - Binding the Templating Result to the Smooks bean context:

```
<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>
```

(*See full example in the split-transform-route-jms tutorial*)

Using **<ftl:outputTo>**, you can direct Smooks to write the templating result directly to an **OutputStreamResource**. This is another useful mechanism for splitting (http://www.smooks.org/mediawiki /index.php?title=V1.2:Smooks_v1.2_User_Guide#Splitting_.26_Routing) huge messages into smaller (more consumable) messages that can then be processed individually.

Example - 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-1.2.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:destinationDirectoryPattern>
        <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>
```

(*See full example in the split-transform-route-jms tutorial*)

FreeMarker Transforms using NodeModels

The easiest way to construct message transforms in FreeMarker is to use FreeMarker's NodeModel (http://freemarker.org/docs/xgui_expose_dom.html) facility. This is where FreeMarker uses a W3C DOM as the Templating model, referencing the DOM nodes directly from inside the FreeMarker template.

Smooks adds two additional capabilities here:

1. The ability to perform this on a fragment basis i.e. you don't have to use the full message as the DOM model, just the targeted fragment.
2. The ability to use NodeModel (http://freemarker.org/docs/xgui_expose_dom.html) in a streaming filter process (see Mixing DOM and SAX Models with Smooks (http://www.smooks.org/mediawiki /index.php?title=V1.2:Smooks_v1.2_User_Guide#Mixing_DOM_and_SAX)).
3. The ability to use it on non XML messages (CSV, EDI etc).

To use this facility in Smooks, you need to define an additional resource that defines/declares the NodeModels to be captured (created in the case of SAX streaming):

```
<?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-1.2.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

    <!--
    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>
```

(*See full example in the xml-to-xml tutorial*)

FreeMarker and the Javabean Cartridge

FreeMarker NodeModel is very powerful and easy to use. The tradeoff is obviously that of performance. Constructing W3C DOMs is not cheap. It also may be the case that the required data has already been extracted and populated into a Java Object model anyway e.g. where the data also needs to be routed to a a JMS endpoint as

Java Objects.

In situations where using the NodeModel is not practical, Smooks allows you to use the Javabean Cartridge to populate a proper Java Object Model (or a Virtual Model). This model can then be used in the FreeMarker Templating process. See the docs on the Javabean Cartridge (http://www.smooks.org/mediawiki /index.php?title=V1.2:Smooks_v1.2_User_Guide#Java_Binding) for more details.

Example (using a Virtual 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-1.2.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>
```

(*See full example in the split-transform-route-file example*)

Programmatic Configuration

FreeMarker templating configurations can be programmatically added to a Smooks instance simply by configuring and adding a FreeMarkerTemplateProcessor (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/templating /org/milyn/templating/freemarker/FreeMarkerTemplateProcessor.html) instance to the Smooks instance. The following example configures a Smooks instance with a Java Binding configuration and a FreeMarker templating configuration:

```
Smooks smooks = new Smooks();

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

// And then just use Smooks as normal... filter a Source to a Result etc...
```

XSL Templating

Configuring XSL templates in Smooks is almost identical to that of configuring FreeMarker templates (http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#FreeMarker_Templating) . It is done through the http://www.milyn.org/xsd/smooks/xsl-1.1.xsd configuration namespace. Just configure this XSD into your IDE and you're in business!

Example:

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

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

</smooks-resource-list>
```

As with FreeMarker, external templates can be configured via URI reference in the <xsl:template> element.

As already stated, configuring XSL templates in Smooks is almost identical to that of configuring FreeMarker templates (See above). For this reason, please consult the FreeMarker configuration docs. Translating to XSL equivalents is simply a matter of changing the configuration namepace. Please read the following sections however.

Points to Note Regarding XSL Support

1. It does not make sense to use Smooks for executing XSLT, unless:
   - You need to perform a fragment transform i.e. you are not transforming the whole message.
   - You need to use other Smooks functionality to perform other operations on the message, such as message splitting, persistence etc.
2. 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 SAX based application of XSL.
3. Smooks applies XSLs on a message fragment basis (i.e. DOM Element Nodes) Vs to the whole document (i.e. DOM Document Node). This can be very useful for fragmenting/modularizing your XSLs, but don't assume that an XSL written and working standalone (externally to Smooks and on the whole document) will automatically work through Smooks without modification. For this reason, Smooks does handle XSLs targeted at the document root node differently in that it applies the XSL to the DOM Document Node (Vs the root DOM Element). The basic point here is that if you already have XSLs and are porting them to Smooks, you may need to make some tweaks to the Stylesheet.
4. XSLs typically contain a template matched to the root element. Because Smooks applies XSLs on a fragment basis, matching against the "root element" is no longer valid. You need to make sure the Stylesheet contains a template that matches against the context node (i.e. the targeted fragment).

My XSLT Works Outside Smooks, but not Inside?

This can happen and is most likely going to be a a result of one of the following:

1. The Fragment based Processing Model: Your Stylesheet contains a template that's using an absolute path reference to the document root node. This will cause issues in the Smooks Fragment based Processing Model because the element being targeted by Smooks is not the document root node. Your XSLT needs to contain a template that matches against the context node being targeted by Smooks.
2. SAX Vs DOM Processing: You are not comparing like with like. Smooks currently only supports a DOM based processing for XSL. In order to do an accurate comparison, you need to use a DOMSource (namespace aware) when executing the XSLT outside Smooks. It has been noticed that a given XSL Processor does not always produce the same output when applying a given XSLT using SAX or DOM.

Groovy Scripting

Support for Groovy (http://groovy.codehaus.org) based scripting is made available through the http://www.milyn.org/xsd/smooks/groovy-1.1.xsd configuration namespace. This adds support for DOM or SAX based Groovy scripting.

Example configuration:

```
<?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/

    <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>
```

Usage Tips

1. **Imports**: Imports can be added via the "imports" element. A number of classes are automatically imported:
   - org.milyn.xml.DomUtils
   - org.milyn.javabean.repository.BeanRepository
   - org.w3c.dom.*
   - groovy.xml.dom.DOMCategory, groovy.xml.dom.DOMUtil, groovy.xml.DOMBuilder
2. **Visited Element**: The visited element is available to the script through the variable "**element**". It is also available under a variable name equal to the element name, but only if the element name contains alpha-numeric characters only.
3. **Execute Before/After**: By default, the script is executed on the visitAfter event. You can direct it to be executed on the visitBefore by setting the "**executeBefore**" attribute to "true".
4. **Comment/CDATA Script Wrapping**: If the script contains special XML characters, it can be wrapped in an XML Comment or CDATA section.

Mixed DOM and SAX with Groovy

Because Groovy (http://groovy.codehaus.org) has a number of very useful DOM processing features, we added support for the mixed DOM and SAX processing models.

What this means is that you can use Groovy's DOM utilities to process the targeted message fragment. The "**element**" received by the Groovy script will be a DOM Element, even when using the SAX filter. This makes Groovy scripting via the SAX filter a lot easier, while at the same time maintaining the ability to process huge messages in a streamed fashion.

Mixed SAX and DOM Gotchas

- Only available in default mode i.e. when "**executeBefore**" equals "**false**". If "**executeBefore**" is configured "**true**", this facility is not available and the Groovy script will only have access to the element as a SAXElement.
- To write the DOM fragment to a Smooks.filterSource StreamResult, "writeFragment" must be called. See example below.
- There is an obvious performance overhead incurred using this facility (DOM construction). That said, it can still be used to process huge messages because of how the DomModelCreator works for SAX. So, it can still process huge message, but it might take a little longer. The tradeoff is usability Vs performance.

Mixed DOM and SAX Example

Take an XML message such as:

```xml
<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>
```

Using Groovy, we want to modify the "supplies" category in the shopping list, adding 2 to the quantity, where the item is "Pens". To do this, we write a simple little Groovy script and target it at the <category> elements in the message. The script simple iterates over the <item> elements in the category and increments the quantity by 2, where the category type is "supplies" and the item is "Pens":

```
<?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/

    <params>
        <param name="stream.filter.type">SAX</param>
    </params>

    <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>
```

Processing Non-XML Data

Smooks relies on a "Stream Reader" for generating a stream of SAX events from the Source message data stream. A Stream Reader is a class that implements the XMLReader interface (http://java.sun.com/j2se/1.5.0/docs/api/org/xml /sax/XMLReader.html) (or the SmooksXMLReader interface (http://www.milyn.org/javadoc/v1.2/smooks /org/milyn/xml/SmooksXMLReader.html)).

By default, Smooks uses the default XMLReader (XMLReaderFactory.createXMLReader() (http://java.sun.com /j2se/1.5.0/docs/api/org/xml/sax/helpers/XMLReaderFactory.html#createXMLReader%28%29)), but can be easily configured to read non-XML data Sources by configuring a specialized XMLReader:

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

    <reader class="com.acme.ZZZZReader" />

    <!--
    Other Smooks resources, e.g. <jb:bean> configs for
    binding data from the ZZZZ data stream into Java Objects....
    -->

</smooks-resource-list>
```

The reader can also be configured with a set of handlers, features and parameters. Here is a full example configuration.

```
<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>
```

A number of non-XML Readers are available with Smooks out of the box:

1. CSVReader (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/csv/org/milyn/csv/CSVReader.html)
2. EDIReader (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/edi/org/milyn/smooks/edi/EDIReader.html)
3. JSONReader (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/edi/org/milyn/json/JSONReader.html)
4. XStreamXMLReader (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/edi/org/milyn/delivery/java/XStreamXMLReader.html)

Any of the above XMLReaders can be configured as outlined above, but some of them have a specialized configuration namespaces that simplify configuration.

Processing CSV

CSV processing through the VCSV Reader is configured through the http://www.milyn.org/xsd/smooks/csv-1.2.xsd configuration namespace.

A simple/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/smook

    <!--
    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>
```

The above configuration will generate an event stream of the form:

```
<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>
```

The <csv-set> and <csv-record> element names can be modified by setting the **rootElementName** and **recordElementName** attributes.

Ignoring Fields

One or more fields of a CSV record can be ignored by specifying the **$ignore$** token in the fields configuration value. You can specify the number of fields to be ignored simply by following the $ignore$ token with a number e.g. "$ignore$3" to ignore the next 3 fields. "$ignore$+" ignores all 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/smook

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

</smooks-resource-list>
```

Binding CSV Records to Java

Smooks v1.2 has added support for making the binding of CSV records to Java Objects a very trivial task. You no longer need to use the Javabean Cartridge directly (i.e. Smooks main Java binding functionality).

A Persons CSV record set such as:

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

Can be bound to a Person of (no getters/setters):

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

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

Using a config of the form:

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

    <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>
```

To execute this configuration:

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

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

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

Smooks also supports creation of 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/smook

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

</smooks-resource-list>
```

The above configuration would produce a Map of Person instances, keyed by the "firstname" value of each Person. It would be executed as follows:

```
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 have the CSV field values bound into Map instances, which are in turn added to a List or a Map.

Programmatic Configuration

Programmatically configuring the CSV Reader on a Smooks instance is trivial (i.e. no XML required). A number of options are available.

Configuring Directly on the Smooks Instance

The following code configures a Smooks instance with a CSVReader for reading a people record set (see above), binding the record set into 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");
```

Of course configuring the Java Binding is totally optional. The Smooks instance could instead (or in conjunction with) be programmatically configured with other Visitor implementations for carrying out other forms of processing on the CSV record set.

CSV List and Map Binders

If you're just interested in binding CSV Records directly onto a List or Map of a Java type that reflects the data in your CSV records, then you can use the CSVListBinder or CSVMapBinder classes.

CSVListBinder:

```
// 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 the lower level APIs:

- Configuring Directly on the Smooks Instance
- Java Binding

Processing EDI

EDI processing in Smooks supported through the http://www.milyn.org/xsd/smooks/edi-1.2.xsd configuration namespace.

The following is a simple/basic configuration:

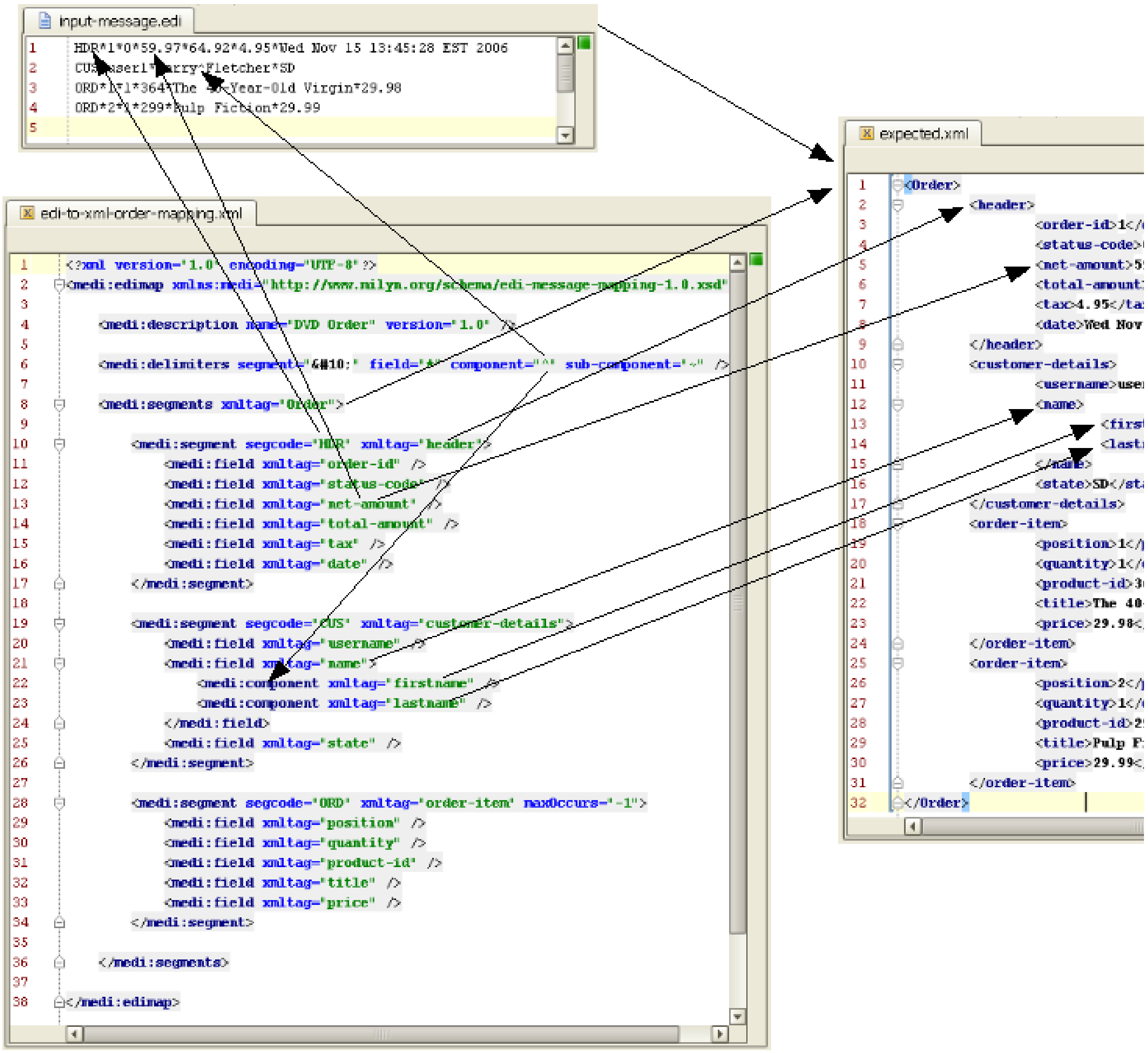
```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd" xmlns:edi="http://www.milyn.org/xsd/smook
        <!--
    Configure the EDI Reader to parse the message stream into a stream of SAX events.
    -->
    <edi:reader mappingModel="edi-to-xml-order-mapping.xml"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

- **mappingModel**: Defines the EDI Mapping Model configuration for processing the EDI message stream to a stream of SAX events that can be processed by Smooks.
- **validate**: This attribute turns on/off datatype validation in the EDI Parser. Validation is on by default. It makes sense to turn datatype validation off on the EDI Reader if the EDI data is being bound into a Java Object model (using Java Bindings ala <jb:bean>). This is because the validation will be happening anyway at the binding level.

EDI Mapping Models

The EDI to SAX Event mapping is performed based on an "Mapping Model" supplied to the EDI Reader. This model must be based on the http://www.milyn.org/xsd/smooks/edi-1.2.xsd schema. From this schema, you can see that segment groups are supported (nested segments), including groups within groups, repeating segments and repeating segment groups. Be sure to review the schema.

The following illustration attempts to create a visualisation of the mapping process. The "input-message.edi" file specifies the EDI input, "edi-to-xml-order-mapping.xml" describes how to map that EDI message to SAX events and "expected.xml" illustrates the XML event stream that would result from applying the mapping.



```
HDR*1*0*59.97*64.92*4.95*Wed Nov 15 13:45:28 EST 2006
CUS*user1*Harry^Fletcher*SD
ORD*1*1*364*The 40-Year-Old Virgin*29.98
ORD*2*1*299*Pulp Fiction*29.99
```

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

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

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

    <medi:segments xmltag="Order">

        <medi:segment segcode="HDR" xmltag="header">
            <medi:field xmltag="order-id" />
            <medi:field xmltag="status-code" />
            <medi:field xmltag="net-amount" />
            <medi:field xmltag="total-amount" />
            <medi:field xmltag="tax" />
            <medi:field xmltag="date" />
        </medi:segment>

        <medi:segment segcode="CUS" xmltag="customer-details">
            <medi:field xmltag="username" />
            <medi:field xmltag="name">
                <medi:component xmltag="firstname" />
                <medi:component xmltag="lastname" />
            </medi:field>
            <medi:field xmltag="state" />
        </medi:segment>

        <medi:segment segcode="ORD" xmltag="order-item" maxOccurs="-1">
            <medi:field xmltag="position" />
            <medi:field xmltag="quantity" />
            <medi:field xmltag="product-id" />
            <medi:field xmltag="title" />
            <medi:field xmltag="price" />
        </medi:segment>

    </medi:segments>

</medi:edimap>
```

```
<Order>
        <header>
                <order-id>1</
                <status-code>
                <net-amount>5
                <total-amount
                <tax>4.95</ta
                <date>Wed Nov
        </header>
        <customer-details>
                <username>use
                <name>
                        <firs
                        <last
                </name>
                <state>SD</st
        </customer-details>
        <order-item>
                <position>1</
                <quantity>1</
                <product-id>3
                <title>The 40
                <price>29.98<
        </order-item>
        <order-item>
                <position>2</
                <quantity>1</
                <product-id>2
                <title>Pulp F
                <price>29.99<
        </order-item>
</Order>
```

So the above illustration attempts to highlight the following:

1. How the message delimiters (segment, field, component and sub-component) are specified in the mapping. In particular, how special characters like the linefeed character are specified using XML Character References.
2. How segment groups (nested segments) are specified. In this case the first 2 segments are part of a group.
3. How the actual field, component and sub-component values are specified and mapped to the target SAX events (to generate the XML).

Segment Cardinality

What's not shown above is how the <medi:segment> element supports the 2 optional attributes "minOccurs" and "maxOccurs" (default value of 1 in both cases). These attributes can be used to control the optional and required characteristics of a segment. A maxOccurs value of -1 indicates that the segment can repeat any number of times in that location of the EDI message (unbounded).

Segment Groups

Segment groups can be added using the <segmentGroup> element. A Segment group is matched by the first segment in the group. A Segment Group can contain nested <segmentGroup> elements, but the first element in a <segmentGroup> must be a <segment>. <segmentGroup> elements support minOccurs/maxOccurs cardinality. They also support an optional "xmlTag" attribute, when if present will result in the XML generated by a matched segment group being inserted inside an element having the name of the xmlTag attribute value.

Segment Matching

Segments are matched in one of 2 ways:

1. By an exact match on the segment code (segcode).
2. By a regex pattern match (http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/package-summary.html) on the full segment, where the segcode attribute defines the regex pattern (e.g. segcode="1A*a.*").

Required Values & Truncation

- **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> and <sub-component> can be present in a message in one of the following states:

1. Present with a value *(required="true")*
2. Present without a value *(required="false")*
3. Not Present *(required="false" and truncatable="true")*

Imports

Many message groups use the same segment definitions. Being able to define these segments once and import them into a top level configuration saves on a lot of duplication. A simple configuration demonstrating the import feature would be as follows:

```
<?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="exa

    <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>
```

The configuration-example above demonstrates the use of import that were introduced in Smooks v1.1, where single segments or segments containing childsegments can be separated into a separate file for better reuse in the future.

- **segref**: Contains a "namespace:name" referencing the segment to import.
- **truncatableSegments**: Overrides the 'truncatableSegments' specified in the imported resource mapping file.
- **truncatableFields**: Overrides the 'truncatableFields' specified in the imported resource mapping file.
- **truncatableComponents**: Overrides the 'truncatableComponets' specified in the imported resource mapping file.

Type Support

Since version 1.2, the <field>, <component> and <sub-component> elements support a "type" attribute that allows datatype specificaton. It actually consists of 2 attributes:

1. **type**: The type attribute specifies the basic datatype.
2. **typeParameters**: The typeParameters attribute specifies data decoding parameters for the DataDecoder (http://www.milyn.org/javadoc/v1.2/commons/org/milyn/javabean/decoders/package-summary.html) associated with the specified type.

The following example shows the type support:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
```

This type system has a number of uses:

- Field Validation.
- EJC - Edifact Java Compiler

Programmatic Configuration

Programmatically configuring the Smooks instance to use the EDIReader is done through the EDIReaderConfigurator (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/edi/org/milyn/smooks/edi/EDIReaderConfigurator.html) :

```
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(....);
```

EJC - Edifact Java Compiler

EJC makes the process of going from EDI to Java much simpler. EJC is similar to JAXBs (http://jaxb.dev.java.net/) XJC, accept it is for EDI messages.

EJC generates:

1. A Java Object model for a given EDI Mapping Model.
2. A Smooks Java Binding config to populate the Java Object model from an instance of the EDI message described by the EDI Mapping Model (see #1 above).
3. A Factory class that makes it very east to use EJC to bind EDI data to Java Object Model.

EJC 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();
```

EJC can be executes through Maven or Ant.

EJC Maven Plugin

Executing the Maven Plugin for EJC is very simple. You just need to install the plugin in your POM file as follows:

```
<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>
```

The plugin has 3 configuration parameters:

- **ediMappingFile**: The path to the EDI Mapping Model File, within the maven project. (**Optional** - default "src/main/resources/edi-model.xml").
- **packageName**: The Java package into which the generated Java Artifacts are to be located (Java Object Model and Factory class).
- **destDir**: The destination directory in which the generated artifacts are created and compiled from. (**Optional** - default "target/ejc").

EJC Ant Task

Executing EJC from an Ant script is trivial. Just configure the EJC Ant task and execute it:

```
<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>
```

Using EJC

The easiest way to get going with EJC is to check out the EJC Example.

Processing JSON

Processing JSON with Smooks requires a JSON reader to be configured:

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

    <json:reader/>

</smooks-resource-list>
```

The following example demonstrates key replacement:

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

    <json:reader>
        <json:keyMap>
            <json:key from="some key">someKey</json:key>
            <json:key from="some&amp;key" to="someAndKey" />
        </json:keyMap>
    </json:reader>

</smooks-resource-list>
```

The following is a full configuration example for the 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/smoo

    <json:reader keyWhitspaceReplacement="_" keyPrefixOnNumeric="n" illegalElementNameCharReplacement="." nullV

</smooks-resource-list>
```

- **keyWhitspaceReplacement**: The replacement character for whitespaces in a json map key. By default this not defined, so that the reader doesn't search for whitespaces.
- **keyPrefixOnNumeric**: The prefix character to add if the JSON node name starts with a number. By default

this is not defined, so that the reader doesn't 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.
- **nullValueReplacement**: The replacement string for JSON NULL values. Default is an empty string.
- **encoding**: The encoding of the input stream. Default of 'UTF-8'

Programmatic Configuration

Smooks is programmatically configured to read a JSON configuration using the JSONReaderConfigurator (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/json/org/milyn/json/JSONReaderConfigurator.html) class.

```
Smooks smooks = new Smooks();

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

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

Configuring the Default Reader

To set features on the default reader, simply omit the class name from the configuration:

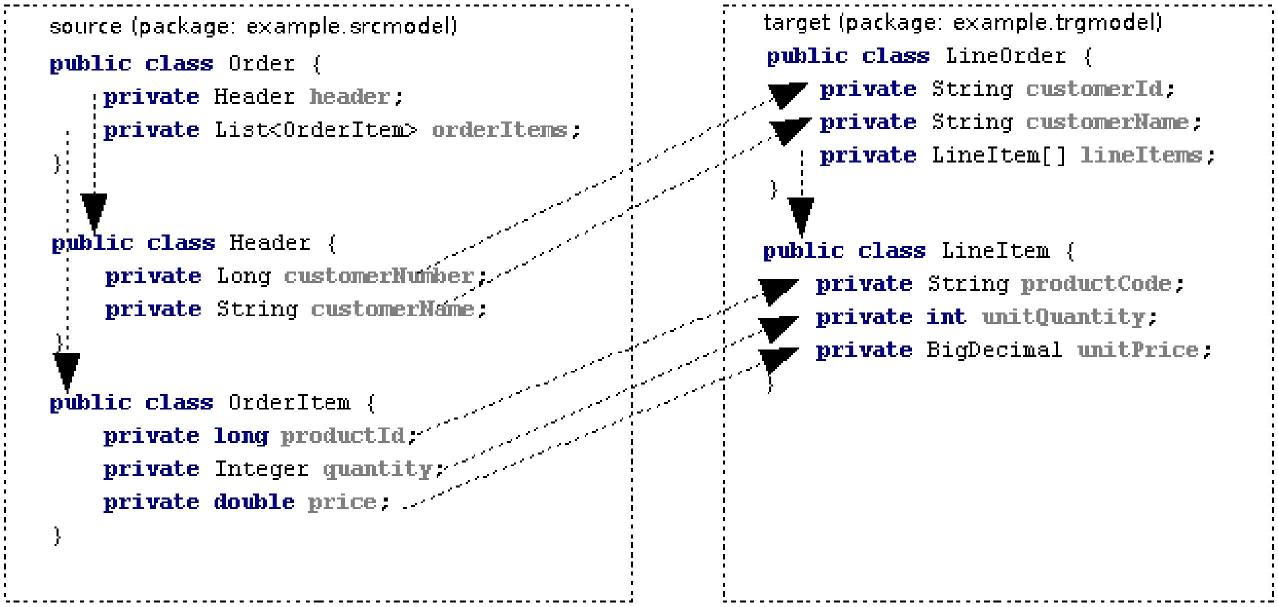
```
<reader>
    <features>
        <setOn feature="http://a" />
        <setOn feature="http://b" />
        <setOff feature="http://c" />
        <setOff feature="http://d" />
    </features>
</reader>
```

Java to Java Transformations

Smooks can transform one Java object graph to another Java object graph. For this transformation Smooks uses the SAX processing model, which means no intermediate object model is constructed for populating the target Java object graph. Instead, we go straight from the source Java object graph, to a stream of SAX events, which are used to populate the target Java object graph.

Source and Target Object Models

The required mappings from the source to target Object models are as follows:



```
source (package: example.srcmodel)
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;
}
```

```
target (package: example.trgmodel)
public class LineOrder {
    private String customerId;
    private String customerName;
    private LineItem[] lineItems;
}

public class LineItem {
    private String productCode;
    private int unitQuantity;
    private BigDecimal unitPrice;
}
```

Source Model Event Stream

Using the Html Smooks Report Generator tool, we can 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>
```

So we need to target the Smooks Javabean resources at this event stream. This is shown in the Smooks Configuration.

Smooks Configuration

The Smooks configuration for performing this transform ("smooks-config.xml") is as follows (see the Source Model Event Stream above):

```
<?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

    <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>
```

Smooks Execution

The source object model is provided to Smooks via a **org.milyn.delivery.JavaSource** Object. This object is created by passing the constructor the root object of the source model. The resulting JavaSource object is used in the **Smooks#filter** method. The resulting code could look like as follows:

```
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");
}
```

Rules

Rules in Smooks refer to a general concept and is not specific to any cartridge. A RuleProvider can be configured and referenced from other components. As of Smooks v1.2, the only Cartridge using Rules functionality is the Validation Cartridge.

So, lets start by looking at what rules in Smooks are, and how they are used.

Rule configuration

Rules are centrally defined through "ruleBase" definitions. A single Smooks config can reference many "ruleBase" definitions. A rulesBase configuration as a **name**, a rule **src** and a rule **provider**. The format of the rule source ("src") is entirely dependent on the provider implementation. The only requirement is that the individual rules be named (unique within the context of a single source) so as they can be referenced by their name.

An example of a ruleBase configuration is as follows:

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

    <rules:ruleBases>
        <rules:ruleBase name="regexAddressing" src=""/org/milyn/validation/address.properties" provider="org.mi
        <rules:ruleBase name="order" src="/org/milyn/validation/order/rules/order-rules.csv" provider="org.mily
    </rules:ruleBases>

</smooks-resource-list>
```

Rulebase Configuration Options

The following are the configuration options for the <rules:ruleBase> configuration element.

- **name**: Is used to reference this rule from other components, like from a validation configuration that we will look at shortly. Required.
- **src**: Is a file or anything meaningful to the RuleProvider. This could be a file containing rules for example. Required.
- **provider**: Is the actual provider implementation that you want to use. This is where the different technologies come into play. In the above configuration we have one RuleProvider that uses regular expression. As you might have guessed you can specify multiple ruleBase element and have as many RuleProviders you need. Required.

RuleProvider Implementations

Rule Providers implement the **org.milyn.rules.RuleProvider** interface.

Smooks v1.2 supports 2 RuleProvider implementations out of the box:

1. RegexProvider
2. MVELProvider

You can easily create custom RuleProvider implementations. Future versions of Smooks will probably include support for e.g. a Drools RuleProvider.

Regex Provider

As it's name suggests, the RegexProvider is based on regular expression. It allows you to define low level rules specific to the format of specific fields of data in the message being filtered e.g. that a particular field is a valid email address.

Configuration of a Regex ruleBase would look like this:

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

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

</smooks-resource-list>
```

Regex expressions are defined in standard .properties file format. An example of a "customer.properties" Regex rule definition file (from the above example) might be as follows:

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

Useful Regular Expressions

See Useful Regular Expressions.

MVELProvider

The MVEL (http://mvel.codehaus.org) Provider allows rules to be defined as MVEL expressions. These expressions are executed on the contents of the Smooks Javabean bean context. That means they require Data to be bound (from the message being filtered) into Java objects in the Smooks bean context. This allows you to define more complex (higher level) 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".

Note: *Be sure to read the section on Java Binding*.

Configuration of an MVEL ruleBase would look like this:

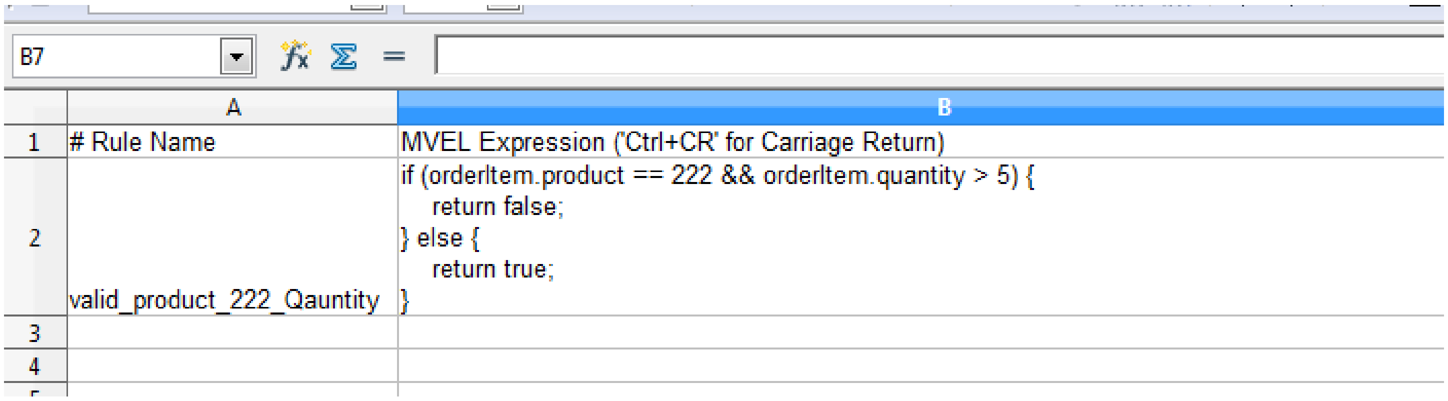
```
<?xml version="1.0"?>
<smooks-resource-list  xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"  xmlns:rules="http://www.milyn.org/xsd/s

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

</smooks-resource-list>
```

MVEL rules must be defined as Comma Separated Value (CSV) files. The easiest way to edit these files is through a Spreadsheet Application (e.g. OpenOffice or Excel). Each rule record contains 2 fields:

1. A Rule Name
2. An MVEL Expression

Comment/header rows can be added by prefixing the first field with a hash ('#') character.

An example of an MVEL rule CSV file as seen in OpenOffice is as follows:


B7

	A	B
1	# Rule Name	MVEL Expression ('Ctrl+CR' for Carriage Return)

2	valid_product_222_Qauntity	if (orderItem.product == 222 && orderItem.quantity > 5) {
		    return false;
		} else {
		    return true;
		}

3		
4		

Validation

The Smooks Validation Cartridge builds on the functionality provided by the Rules Cartridge, to provide Rules based fragment validation.

The type of validation provided by the components of the Smooks Validation Cartridge allows you to perform more detailed validation (over the likes of XSD/Relax) on message fragments. As with everything in Smooks, the Validation functionality is supported across all supported data formats. This means you can perform strong validation on not just XML data, but also on EDI, JSON, CSV etc.

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

Validation configuration

Smooks supports a number of different Rule Provider types that can be used by the Validation Cartridge. They provide different levels of validation. These different forms of Validation are configured in exactly the same way. The Smooks Validation Cartridge sees a Rule Provider as an abstract resource that it can target at message fragments in order to perform validation on the data in that message fragment.

A Validation rule configuration is very simple. You simply need to specify:

- **executeOn**: The fragment on which the rule is to be executed.
- **excecuteOnNS**: The fragment namespace (NS) that that 'executeOn' belongs to.
- **name**: The name of the rule to be applied. This is a Composite Rule Name that references a ruleBase and ruleName combination in a dot delimited format i.e. "ruleBaseName.ruleName".
- **onFail**: The severity of a failed match for the Validation rule. See onFail section for details.

An example of a Validation rule configuratio0n would be as follows:

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

Configuring max failures

One can set a maximum number of validation failures per Smooks filter operation. An exception will be thrown if this max value is exceeded. Note that validations configured with **OnFail.FATAL** will always throw an exception and stop processing.

To configure the maximum validation failures add this following to you Smooks configuration:

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

onFail

The onFail attribute in the validation configuration specified what action should be taken when a rule matches. This is all about reporting back valdiation failures.

The following options are available:

- **OK'**: Save the validation as an ok validation. Calling ValidationResults.getOks will return all validation warnings. This can be useful for content based routing.
- **WARN**: Save the validation as a warning. Calling ValidationResults.getWarnings will return all validation warnings.
- **ERROR**: Save the validation as an error. Calling ValidationResults.getErrors will return all validation errors.
- **FATAL**: Will throw a ValidationException as soon as a validation failure occurs. Calling ValidationResults.getFatal will return the fatal validation failure.

Composite Rule Name

When a RuleBase is references in Smooks you use a composite rule name in the following format:

```
<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/resource.

Validation Results

Validation results are captured by the Smooks.filterSource by specifying a ValidationResult instance in the filterSource method call. When the filterSource method returns, the ValidationResult instance will contain all validation data.

An example of executing Smooks in order to perform message fragment validation is as follows:

```
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 from the above code, individual warning, error etc validation results are made available from the ValidationResult object in the form of **OnFailResult** instances. The OnFailResult object provides details about an individual failure.

Localized Validation Messages

The Validation Cartridge provides support for specifying localized messages relating to Validation failures. These messages can be defined in standard Java ResourceBundle files (.properties format). A convention is used here, based on the rule source name ("src"). The validation message bundle base name is derived from the rule source ("src") by dropping the rule source file extension and adding an extra folder named "i18n" e.g. for an MVEL ruleBase source of "/org/milyn/validation/order/rules/order-rules.csv", the corresponding validation message bundle base name would be "/org/milyn/validation/order/rules/i18n/order-rules".

The validation cartridge supports application of **FreeMarker templates** on the localized messages, allowing the messages to contain contextual data from the bean context, as well as data about the actual rule failure. FreeMarker based messages must be prefixed with "ftl:" and the contextual data is references using the normal FreeMarker notation. The beans from the bean context can be referenced directly, while the RuleEvalResult and rule failure path can be referenced through the "ruleResult" and "path" beans.

Example message using RegexProvider rules:

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

Example

See the Validation Example.

Processing Huge Messages (GBs)

One of the main features introduced in Smooks v1.0 is the ability to process huge messages (Gbs in size). Smooks supports the following types of processing for huge messages:

- **One-to-One Transformation**: This is the process of transforming a huge message from it's source format (e.g. XML), to a huge message in a target format e.g. EDI, CSV, XML etc.
- **Splitting & 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 (i.e. in a declarative manner). Typically, any of the above types of processing would have required writing quite a bit of ugly/unmaintainable code. It might also have been implemented as a multi-stage process where the huge message is split into smaller messages (stage #1) and then each smaller message is processed in turn to persist, route etc. (stage #2). This would all be done in an effort to make that ugly/unmaintainable code a little more maintainable and reusable. With Smooks, most of these use-cases can be handled without writing any code. As well as that, 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: *Be sure to read the section on Java Binding*.

Performance Hint

- When processing huge messages with Smooks, make sure you are using the **SAX filter**.

One-to-One Transformation

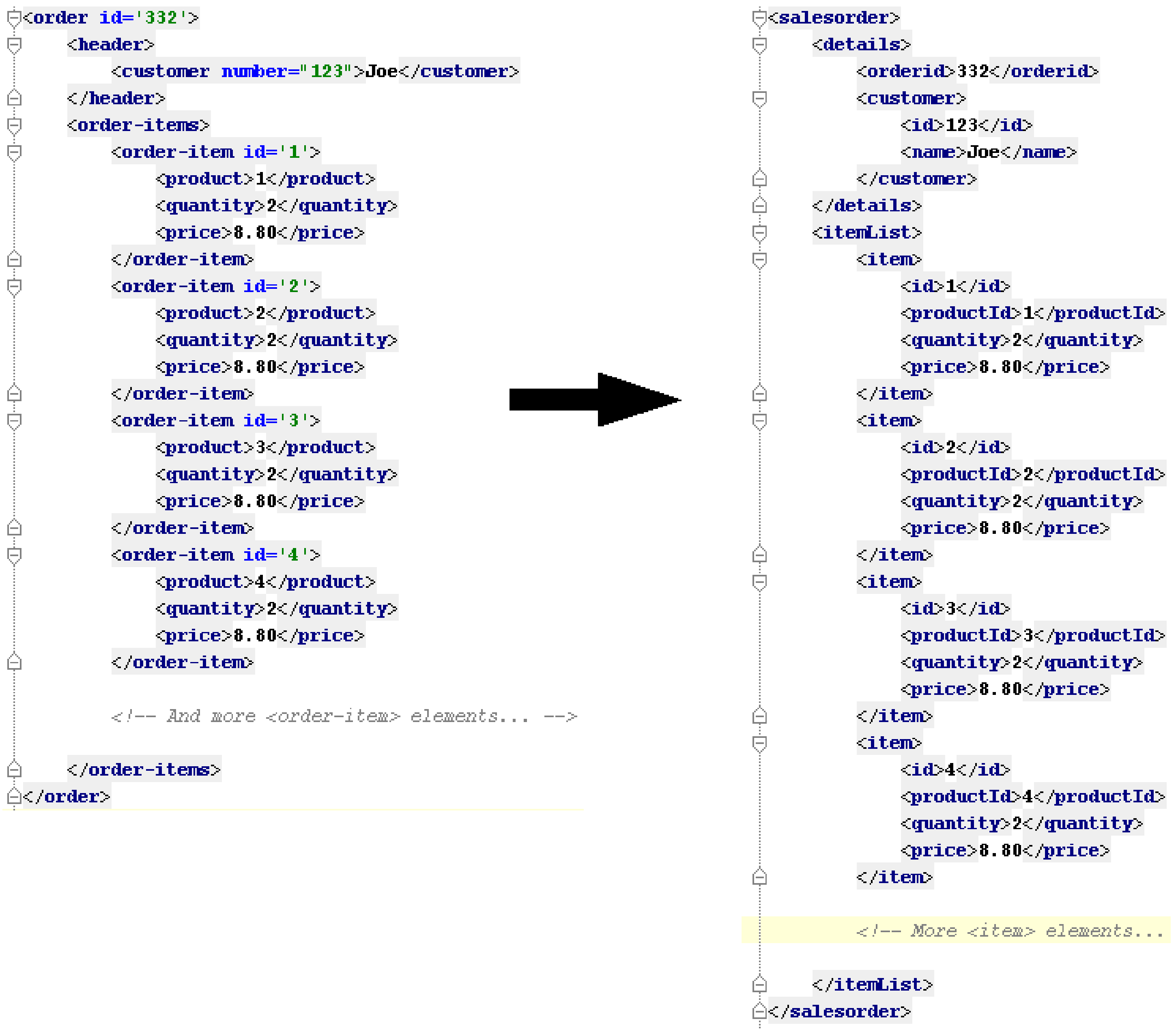
If the requirement is to process a huge message by transforming it into a single message of another format, the easiest mechanism with Smooks is to apply multiple FreeMarker templates to the Source message Event Stream, outputting to a Smooks.filterSource Result stream.

This can be done in one of 2 ways with FreeMarker templating, depending on the type of model that's appropriate:

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

Option #1 above is obviously the option of choice, if the tradeoffs are OK for your use case. Please see the FreeMarker Templating docs for more details.

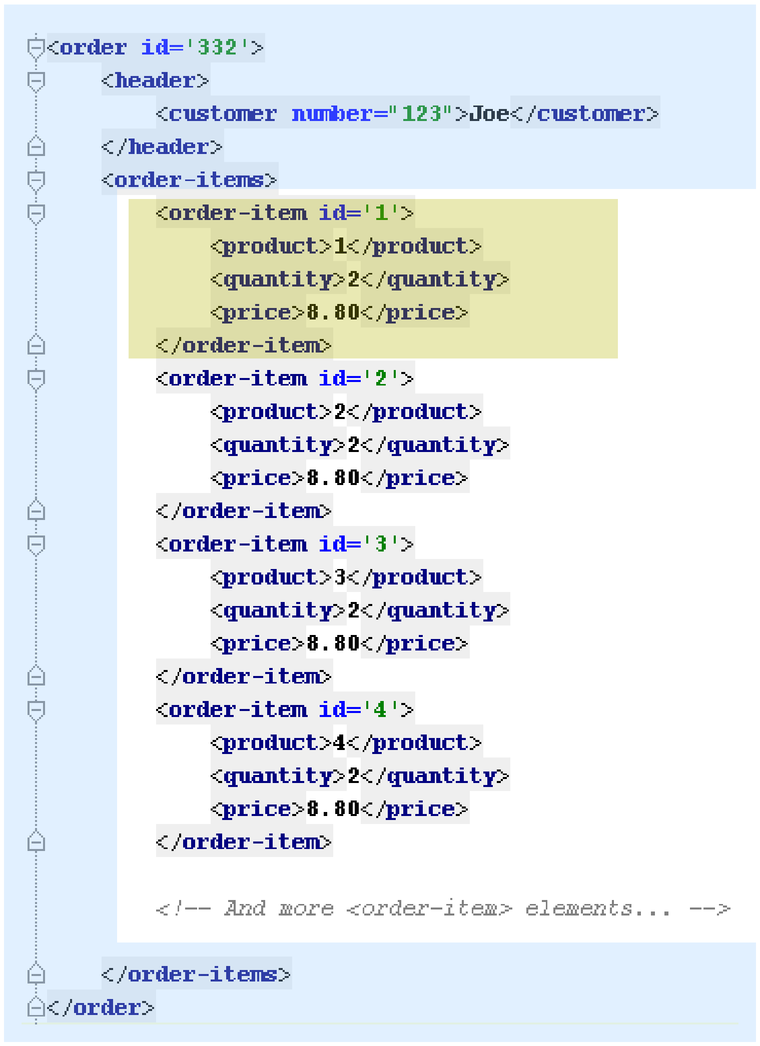
The following images shows an <order> message, as well as the <salesorder> message to which we need to transform the <order> message:



```
<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-item id='4'>
            <product>4</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>

        <!-- And more <order-item> elements... -->

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

```
<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>
        <item>
            <id>2</id>
            <productId>2</productId>
            <quantity>2</quantity>
            <price>8.80</price>
        </item>
        <item>
            <id>3</id>
            <productId>3</productId>
            <quantity>2</quantity>
            <price>8.80</price>
        </item>
        <item>
            <id>4</id>
            <productId>4</productId>
            <quantity>2</quantity>
            <price>8.80</price>
        </item>

        <!-- More <item> elements...

    </itemList>
</salesorder>
```

Imagine a situation where the <order> message contains millions of <order-item> elements. Processing a huge message in this way with Smooks and FreeMarker (using NodeModels) is quite straightforward. Because the message is huge, we need to identify multiple NodeModels in the message, such that the runtime memory footprint is as low as possible. We cannot process the message using a single model, as the full message is just too big to hold in memory. In the case of the <order> message, there are 2 models, one for the main <order> data (blue highlight) and one for the <order-item> data (beige highlight):



```
<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-item id='4'>
            <product>4</product>
            <quantity>2</quantity>
            <price>8.80</price>
        </order-item>

        <!-- And more <order-item> elements... -->

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

So in this case, the 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 (i.e. they are not collected). See Mixing DOM and SAX Models with Smooks.

Configuring Smooks to capture multiple NodeModels for use by the FreeMarker templates is just a matter of configuring the **DomModelCreator** Visitor, targeting it at the root node of each of the models. Note again that Smooks also makes this available to SAX filtering (the key to processing huge message). The Smooks configuration for creating the NodeModels for this message are:

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

    <!--
    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 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... -->
```

Now the FreeMarker templates need to be added. We need to apply 3 templates in total:

1. A template to output the order "header" details, up to but not including the order items.
2. A template for each of the order items, to generate the <item> elements in the <salesorder>.
3. A template to close out the message.

With Smooks, we implement this by defining 2 FreeMarker templates. One to cover #1 and #3 (combined) above, and a seconds to cover the <item> elements.

The first FreeMarker template is targeted at the <order-items> element and looks as follows:

```
<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>
```

You will notice the **<?TEMPLATE-SPLIT-PI?>** Processing Instruction. This 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.

The second FreeMarker template is very straightforward. It simply 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.

And that's it! This is available as a runnable example in the Tutorials section.

This approach to performing a One-to-One Transformation of a huge message works simply 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't work if the transformation is so obscure as to always require full access to all the data in the source message e.g. 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.

Splitting & Routing

Another common approach to processing large/huge messages is to split them out into smaller messages that can be processed independently. Of course Splitting and Routing is not just a solution for processing huge messages. It's often needed with smaller messages too (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 - "Content Base Routing") to different departments or partners for processing. Under these conditions, the message formats required at the different destinations may also vary e.g.

- "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,
- etc etc

With Smooks, all of the above is possible. You can perform multiple splitting and routing operations to multiple destinations (of different types) in a single pass over a message.

The key to processing huge messages is to make sure that you always maintain a small memory footprint. You can do this using the Javabean Cartridge by making sure you're only binding the most relevant message data (into the bean context) at any one time. In the following sections, the examples are all based on splitting and routing of order-items out of an order message. The solutions shown all work for huge messages because the Smooks Javabean Cartridge binding configurations are implemented such that the only data held in memory at any given time is the main order details (order header etc) and the "current" order item details.

Complex splitting operations are supported through use of the Javabean Cartridge to extract the data for the split-message. In this way, you can extract and recombine data from across different sub-hierarchies of the Source message, to produce the split messages. It also means you can (through the use of templating) easily generate the split messages in a range of different formats. More on this later.

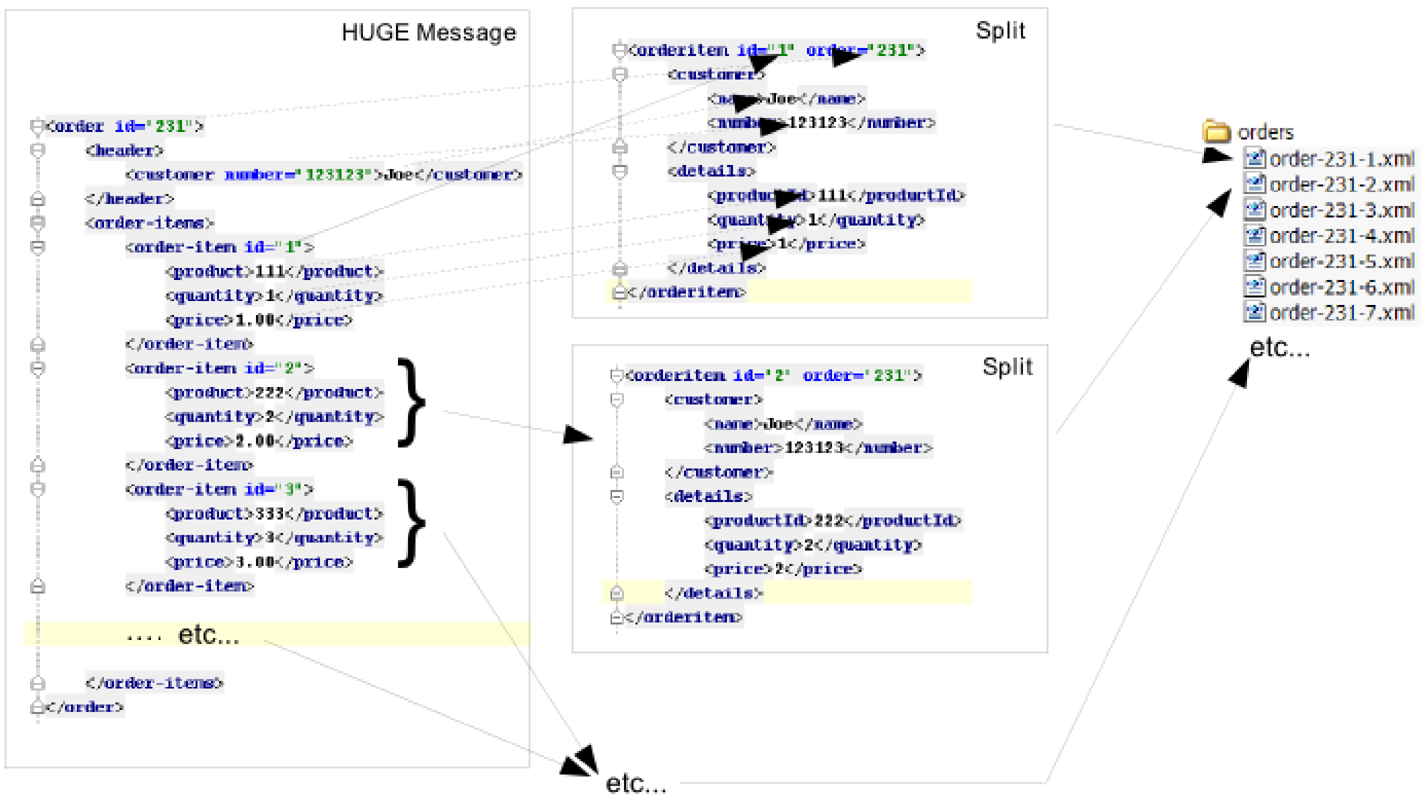
Routing to File

File based routing is performed via the the **<file:outputStream>** configuration from the http://www.milyn.org /xsd/smooks/file-routing-1.1.xsd configuration namespace.

This section illustrates how you can combine the following Smooks functionality to split a message out into smaller messages on the file system.

1. The **Javabean Cartridge** for extracting data from the message and holding it in variables in the bean context. In this case, we could also use DOM NodeModels for capturing the order and order-item data to be used as the templating data models.
2. The **<file:outputStream>** configuration from the **Routing Cartridge** for managing file system streams (naming, opening, closing, throttling creation etc).
3. The **Templating Cartridge** (FreeMarker Templates) for generating the individual split messages from data bound in the bean context by the Javabean Cartridge (see #1 above). The templating result is written to the file output stream (#2 above).

In the example, we want to process a huge order message and route the individual order item details to file. The following illustrates what we want to achieve. As you can see, the split messages don't just contain data from the order item fragments. They also contain data from the order header and root elements.



```
<order id='231'>
    <header>
        <customer number="123123">Joe</customer>
    </header>
    <order-items>
        <order-item id="1">
            <product>111</product>
            <quantity>1</quantity>
            <price>1.00</price>
        </order-item>
        <order-item id="2">
            <product>222</product>
            <quantity>2</quantity>
            <price>2.00</price>
        </order-item>
        <order-item id="3">
            <product>333</product>
            <quantity>3</quantity>
            <price>3.00</price>
        </order-item>
        .... etc...
    </order-items>
</order>
```

```
<orderitem id="1" order="231">
    <customer>
        <name>Joe</name>
        <number>123123</number>
    </customer>
    <details>
        <productId>111</productId>
        <quantity>1</quantity>
        <price>1</price>
    </details>
</orderitem>
```

```
<orderitem id='2' order='231'>
    <customer>
        <name>Joe</name>
        <number>123123</number>
    </customer>
    <details>
        <productId>222</productId>
        <quantity>2</quantity>
        <price>2</price>
    </details>
</orderitem>
```

To achieve this with Smooks, we assemble the following Smooks 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-1.2.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...
        -->
        <params>
            <param name="stream.filter.type">SAX</param>
        </params>

        <!-- 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:destinationDirectoryPattern>
            <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 configuration #1 and #2 define the Java Bindings for extracting the order header information (config #1) and the order-item information (config #2). This is the key to processing a huge message; making sure that we 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 #3 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 #4 defines the FreeMarker templating resource used to write the split messages to the OutputStream created by the <file:outputStream> (config #3). See how config #4 references the <file:outputStream> resource. The Freemarker temaplte 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>
```

Routing to JMS

JMS routing is performed via the the **<jms:router (http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd) >** configuration from the http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd configuration namespace.

The following is an example <jms:router (http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd) > configuration that routes an "orderItem_xml" bean to a JMS Queue named "smooks.exampleQueue" (also read the "Routing to File" example):

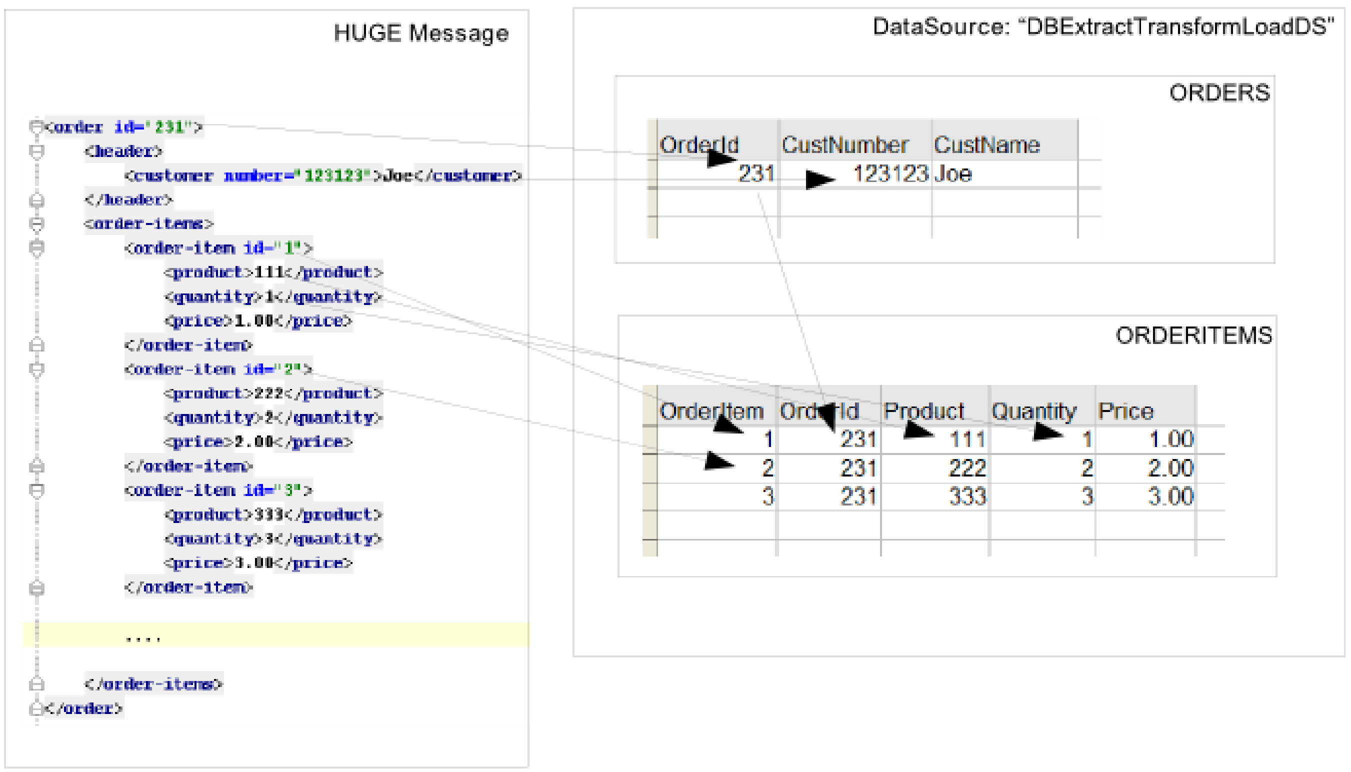
```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jms="http://www.milyn.org/xsd/smooks/jms-routing-1.2.xsd"
                      xmlns:ftl="http://www.milyn.org/xsd/smooks/freemarker-1.1.xsd">

        <!--
        Filter the message using the SAX Filter (i.e. not DOM, so no
        intermediate DOM, so we can process huge messages...
        -->
        <params>
        <param name="stream.filter.type">SAX</param>
        </params>

(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
              -->
              <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 (i.e. as a String). We could also have routed a full Object Model, in which case it would be routed as a Serialized ObjectMessage.

Routing to a Database using SQL

Routing to a Database is also quite easy. Please read the "Routing to File" section above before reading this section.

So we take the same scenario as with the File Routing example above, but this time we want to route the order and order item data to a Database. This is what we want to achieve:



```
<order id='231'>
    <header>
        <customer number="123123">Joe</customer>
    </header>
    <order-items>
        <order-item id="1">
            <product>111</product>
            <quantity>1</quantity>
            <price>1.00</price>
        </order-item>
        <order-item id="2">
            <product>222</product>
            <quantity>2</quantity>
            <price>2.00</price>
        </order-item>
        <order-item id="3">
            <product>333</product>
            <quantity>3</quantity>
            <price>3.00</price>
        </order-item>

        ....

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

OrderId	CustNumber	CustName
231	123123	Joe

OrderItem	OrderId	Product	Quantity	Price
1	231	111	1	1.00
2	231	222	2	2.00
3	231	333	3	3.00

First we need to 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-1.2.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>

</smooks-resource-list>
```

Next we 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.

The Datasource configuration (namespace http://www.milyn.org/xsd/smooks/datasource-1.1.xsd):

```
<?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

    <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>
```

The <db:executor> configurations (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: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.product
    </db:executor>

    <!-- Ignoring updates for now!! -->

</smooks-resource-list>
```

Check out the db-extract-transform-load example.

Message Splitting & Routing

Please refer to the Splitting & Routing section in the previous section.

Persistence (Database Reading and Writing)

There are three methods for reading and writing to a database from within Smooks. As with many other features of Smooks, this capability relies heavily on the Smooks Java Binding capabilities provided in the Javabean Cartridge, or extends it:

- 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.
- Use an entity persistence framework (like Ibatis, Hibernate or any JPA compatible framework) to access a

database and use it's query language or CRUD methods to read from it or write to it. See the section on Entity Persistence Frameworks.
- Use custom Data Access Objects (DAO's) to access a database and use it's CRUD methods to read from it or write to it. Again, see DAO Support.

Note: *Be sure to read the section on Java Binding.*

Entity Persistence Frameworks

With the new Smooks Persistence cartridge in Smooks 1.2, you can directly use several entity persistence frameworks from within Smooks (Hibernate, JPA etc).

Lets take a look at a Hibernate example. The same principals follow for any JPA compliant framework.

The data we are going to process is an XML order message. It should be noted however, that 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!

```
<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>
```

The Hibernate entities are:

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

What we want to do here is to process and persist the <order>. First thing we need to do is to bind the order data into the Order entities (Order, OrderLine and Product). To do this we need to:

1. **Create** and populate the Order and OrderLine entities using the Java Binding framework.
2. **Wire** each OrderLine instance into the Order instance.
3. Into each OrderLine instance, we need to **lookup and wire** in the associated order line Product entity.
4. And finally, we need to **insert** (persist) the Order instance.

To do this, we need the following Smooks configuration:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.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>
```

If we want to use the named query "productById" instead of the query string then the DAO locator configuration will look like this:

```
<dao:locator beanId="product" lookupOnElement="order-item" lookup="product.byId" onNoResult="EXCEPTION" uniqueR
    <dao:params>
        <dao:value name="id" data="product" decoder="Integer"/>
    </dao:params>
</dao:locator>
```

The following code executes Smooks. Note that we use a SessionRegister object so that we can access the Hibernate Session 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();
```

DAO Support

Now let's take a look at a DAO based example. The example will read an XML file containing order information (note that this works just the same for EDI, CSV etc). Using the javabean cartridge, it will bind the XML data into a set of entity beans. Using the id of the products within the order items (the <product> element) it will locate the product entities and bind them to the order entity bean. 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>
```

The following custom DAO will be used 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.

The following custom DAO will be used 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, you should notice the @Lookup and @Param annotation. The @Lookup annotation declares that the ProductDao#findByProductId 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 let's 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 there position.

The Smooks configuration look likes this:

```
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
                      xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.2.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>
```

The following code executes 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 isbe 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();
```

Message Enrichment

Use the Persistence features of Smooks. The queried data will be bound to the bean context (ExecutionContext).

Use the bound query data to enrich your messages e.g. where you are splitting and routing.

TODO!!

Global Configurations

Global configuration settings are, as the name implies, configuration options that can be set once and be applied to all resources in a configuration.

Smooks supports two types of globals, default properties and global parameters:

- **Default Properties**: Specify default values for <resource-config> attributes. These defaults are automatically applied to SmooksResourceConfigurations (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/cdr/SmooksResourceConfiguration.html) when their corresponding <resource-config> does not specify the attribute. More on this in the following section.
- **Global Configuration Parameters**: Every <resource-config> in a Smooks configuration can specify <param> elements for configuration parameters. These parameter values are available at runtime through the SmooksResourceConfiguration (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/cdr/SmooksResourceConfiguration.html) , or are reflectively injected through the **@ConfigParam** annotation. Global Configuration Parameters are parameters that are defined centrally (see below) and are accessible to all runtime components via the ExecutionContext (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/container/ExecutionContext.html) (Vs the SmooksResourceConfiguration (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/cdr/SmooksResourceConfiguration.html)). More on this in the following sections.

Default Properties

Default properties are properties that can be set on the root element of a Smooks configuration and have them applied to all resource configurations in smooks-conf.xml file. For example, if you have a resource configuration file in which all the resource configurations have the same selector value, you could specify a *default-selector=order* to save specifying the selector on on every resource configuration:

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

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

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

<smooks-resource-list>
```

The following default configuration options are available:

- **default-selector**: Selector that will be applied to all resource-config elements in the smooks configuration file, where a selector is not defined.

- **default-selector-namespace**: The default selector namespace, where a namespace is not defined.
- **default-target-profile**: Default target profile that will be applied to all resources in the smooks configuration file, where a target-profile is not defined.
- **default-condition-ref**: Refers to a global condition by the conditions id. This condition is applied to resources that define an empty "condition" element (i.e. <condition/>) that does not reference a globally defined condition.

Global Configuration Parameters

Global properties differ from the default properties in that they are not specified on the root element and are not automatically applied to resources.

Global parameters are specified in a **<params>** element:

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

Global Configuration Parameters are accessible via the ExecutionContext (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/container/ExecutionContext.html) e.g.:

```
public void visitAfter(final Element element, final ExecutionContext executionContext) throws SmooksException {
    String param1 = executionContext.getConfigParameter("xyz.param1", "defaultValueABC");

    ....
}
```

Global Filter Setting Parameters

The following global configuration options are available for configuring Smooks filtering:

- **stream.filter.type** Determines the type of processing model that will be used. Either SAX or DOM. Please refer to Filtering Process Selection for more information about the processing models. Default is DOM.
- **default.serialization.on**: Determines if default serialization should be switched on (default "true"). Default serialization being turned on simply tells Smooks to locate a StreamResult (http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/transform/stream/StreamResult.html) (or DOMResult) in the Result objects provided to the Smooks.filterSource (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/Smooks.html#filterSource(javax.xml.transform.Source,%20javax.xml.transform.Result...)) method and to, by default, serialize all events to that Result. This behavior can be turned off using this global configuration parameter and can be overriden on a per fragment basis by targetting a Visitor implementation at that fragment that takes ownership of the Result writer (in the case of SAX filtering), or simply modifies the DOM (in the case of DOM filtering). As an example of this, see the FreeMarkerTemplateProcessor (http://www.milyn.org/javadoc/v1.2/smooks-cartridges/templating/org/milyn/templating/freemarker/FreeMarkerTemplateProcessor.html) .
- **terminate.on.visitor.exception**: Determines whether an exception should terminate processing (default "true").
- **maintain.element.stack**: Should Smooks maintain a contextual stack of element names from the source event stream (default "true"). This stack effectively allows Visitor logic to "see" where the are in a message

hierarchy. There is a performance overhead associated with this however and sometimes it is not required.
- **close.source**: Close Source instance streams passed to the Smooks.filterSource (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/Smooks.html#filterSource(javax.xml.transform.Source,%20javax.xml.transform.Result...)) method (default "true"). The exception here is System.in, which will never be closed.
- **close.result**: Close Result streams passed to the Smooks.filterSource (http://www.milyn.org/javadoc/v1.2/smooks/org/milyn/Smooks.html#filterSource(javax.xml.transform.Source,%20javax.xml.transform.Result...)) method (default "true"). The exception here is System.out and System.err, which will never be closed.

Performance Tuning

Like with any Software, when configured or used incorrectly, performance can be one of the first things to suffer. Smooks is no different in this regard.

General

- **Cache and Reuse** the Smooks Object. Initialization of Smooks takes some time and therefore it is important that it is reused.
- **Only use the HTMLReportGenerator in development**. When enabled, the HTMLReportGenerator incurs a significant performance overhead and with large message, can even result in OutOfMemory exceptions.
- **If possible, use SAX filtering**. However, you need to check that all Smooks Cartridges in use are SAX compatible. SAX processing is a lot faster than DOM processing and has a consistently small memory footprint. It is mandatory for processing large messages. See the Filtering Process Selection (DOM or SAX?) section.
- **Contextual selectors** can obviously have a negative effect on performance e.g. 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 them for the sake of performance.

Smooks Cartridges

Every cartridge can have its own performance optimization tips.

Javabean Cartridge

- If possible don't use the Virtual Bean Model. Create Beans instead of maps. Creating and adding data to Maps is a lot slower then creating simple POJO's and calling the setter methods.

ESB Integration

Smooks plugins are available for a number of ESBs:

1. JBoss ESB (http://wiki.jboss.org/wiki/Wiki.jsp?page=MessageTransformation) .
2. Mule (http://www.mulesource.org/display/SMOOKS/Home) .
3. Apache Synapse/WS02 (http://esbsite.org/resources.jsp?path=/mediators /upul/Smooks%20Transform%20Mediator) .

Testing

Unit Testing

Unit testing with Smooks is simple:

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

The test case above uses xmlunit (http://xmlunit.sourceforge.net/) .

The following maven dependency was used for xmlunit in the above test:

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

Retrieved from "http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide"