Processing Huge Messages with Smooks

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:

1. **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.
2. **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).
3. **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).

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 the Smooks.filter 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:

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):

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/smooks/freemarker-1.1.xsd">

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

    <!--
    Create 2 NodeModels.  One high level model for the "order"
    (header etc) and then one 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 v1.0, 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.

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-1.1.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:bindings 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:bindings>
(2) <jb:bindings 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:bindings>

    <!-- Create/open a file output stream.  This is writen to by the
freemarker template (below).. -->
(3) <file:outputStream openOnElement="order-item"
resourceName="orderItemSplitStream">

<file:fileNamePattern>order-${order.orderId}-${order.orderItem.itemId}.xml
</file:fileNamePattern>

<file:destinationDirectoryPattern>target/orders</file:destinationDirectory
Pattern>

<file:listFileNamePattern>order-${order.orderId}.lst</file:listFileNamePat
tern>

        <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>** configuration from the http://www.milyn.org/xsd/smooks/jms-routing-1.1.xsd configuration namespace.

The following is an example <jms:router> 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-1.1.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>

(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:cor
relationIdPattern>
        </jms:message>
        <jms:highWaterMark mark="3"/>
    </jms:router>

(3) <ftl:freemarker applyOnElement="order-item">
        <!--
            Note in the template that we need to use the special FreeMarker
variable ".vars"
            because of the hyphenated variable names ("order-item"). See
http://freemarker.org/docs/ref_specvar.html.
        -->
        <ftl:template>/orderitem-split.ftl</ftl:template>
        <ftl:use>
            <!-- Bind the templating result into the bean context, from
where
                 it can be accessed by the JMSRouter (configured above).
-->
            <ftl:bindTo id="orderItem_xml"/>
        </ftl:use>
    </ftl:freemarker>

</smooks-resource-list>
```

In this case, we route the result of a FreeMarker templating operation to the JMS Queue (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

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:

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-1.1.xsd">

    <!-- Extract the order data... -->
    <jb:bindings 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:bindings>

    <!-- Extract the order-item data... -->
    <jb:bindings 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:bindings>

</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/datasource-1.1.xsd">

 <ds:direct
  bindOnElement="$document"
  datasource="DBExtractTransformLoadDS"
  driver="org.hsqldb.jdbcDriver"
  url="jdbc:hsqldb:hsql://localhost:9201/milyn-hsql-9201"
  username="sa"
  password=""
  autoCommit="false" />

</smooks-resource-list>
```

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:statement>
    </db:executor>

    <!-- And insert each orderItem... -->
    <db:executor executeOnElement="order-item"
datasource="DBExtractTransformLoadDS" executeBefore="false">
     <condition>orderExistsRS.isEmpty()</condition>
     <db:statement>INSERT INTO ORDERITEMS VALUES (${orderItem.itemId},
${order.orderId}, ${orderItem.productId}, ${orderItem.quantity},
${orderItem.price})</db:statement>
    </db:executor>

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

</smooks-resource-list>
```

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